P&S Heterogeneous Systems

Collaborative Computing

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich

Fall 2022

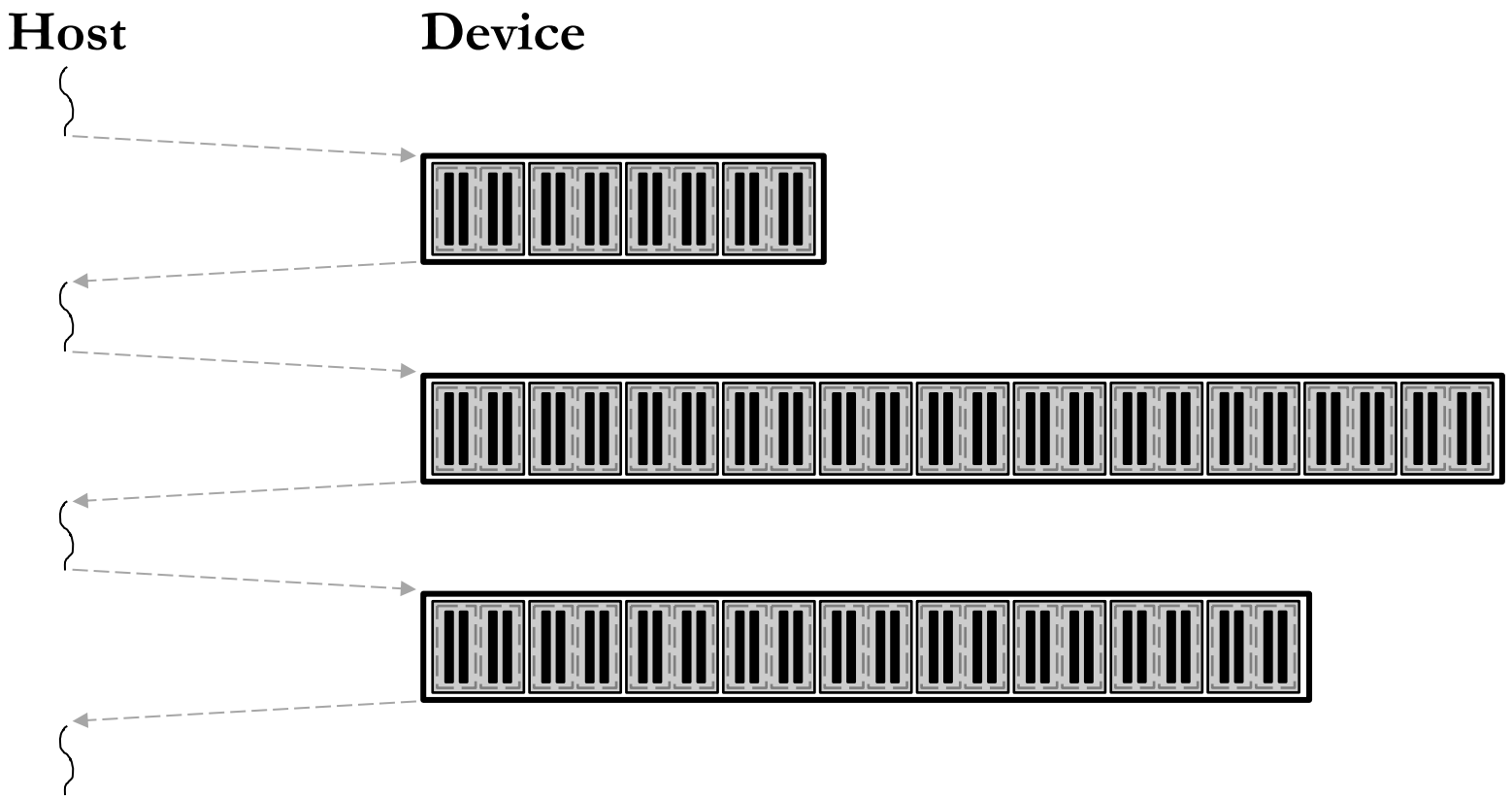
16 January 2023

In Our Previous Lecture…

Dynamic Parallelism

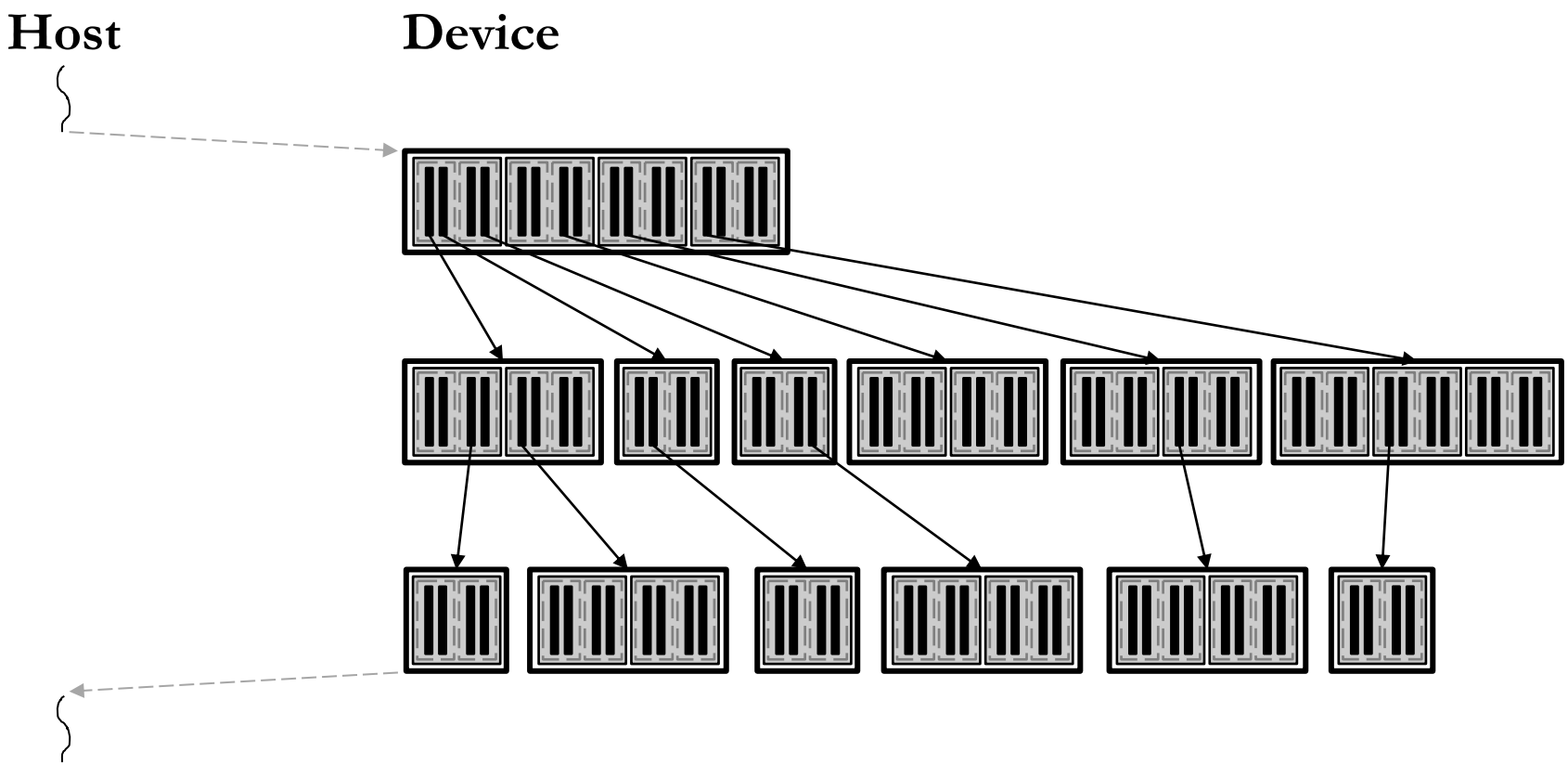
- GPU programming frameworks provide an interface to express dynamic refinement algorithms in a more natural way
  - This dynamic parallelism interface allows GPU threads to launch GPU kernels when new work is dynamically discovered
  - Recall BFS
    - Each node in the frontier has a different number of neighbors
- CUDA Dynamic Parallelism
  - Important semantics when a kernel is launched from a kernel
  - Performance considerations

Kernel Launch without Dynamic Parallelism



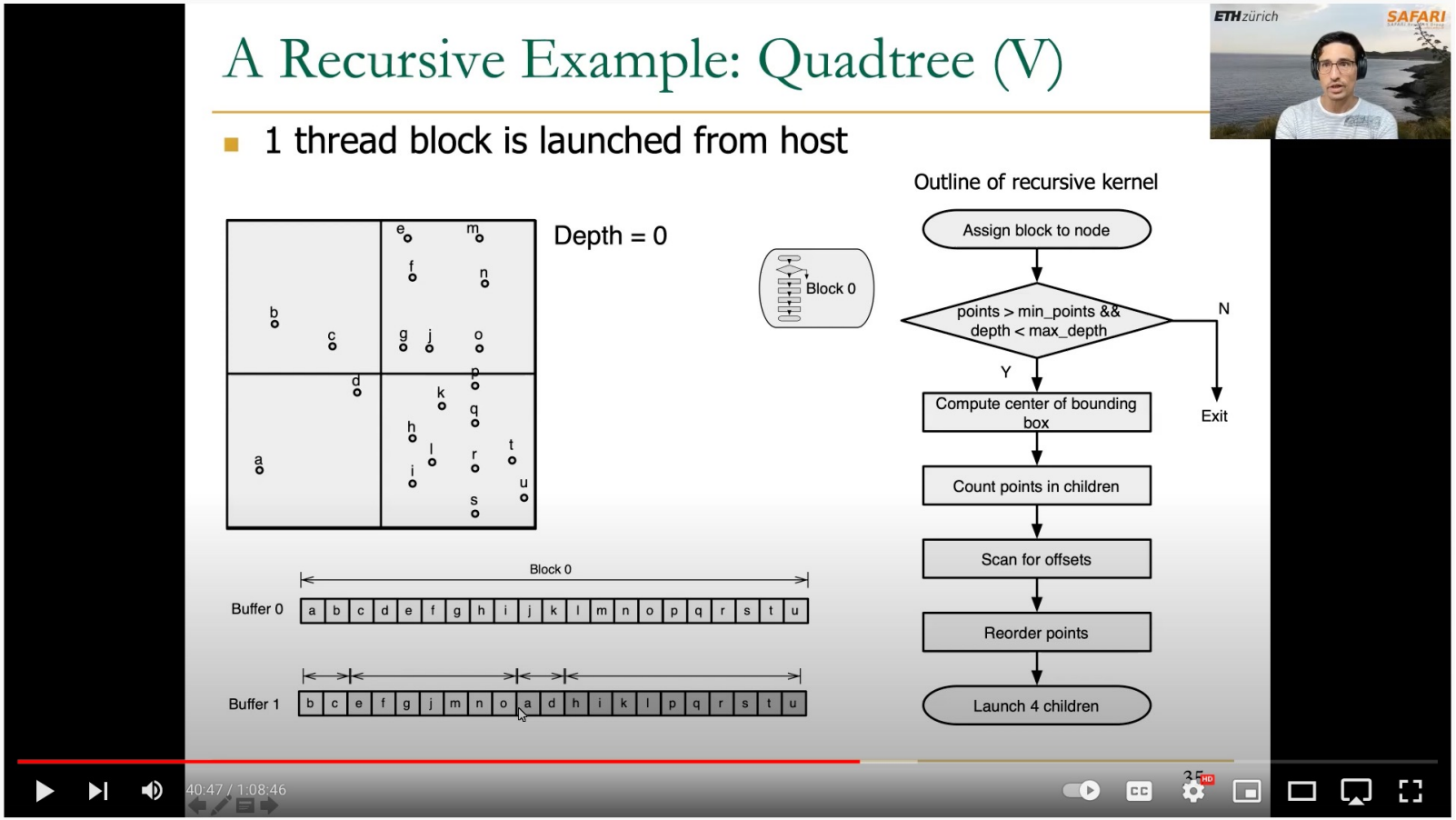
Previously, kernels could only be launched from the host (painful to program!)

Kernel Launch with Dynamic Parallelism



Easier to write programs with dynamically discovered parallelism

Lecture on Dynamic Parallelism

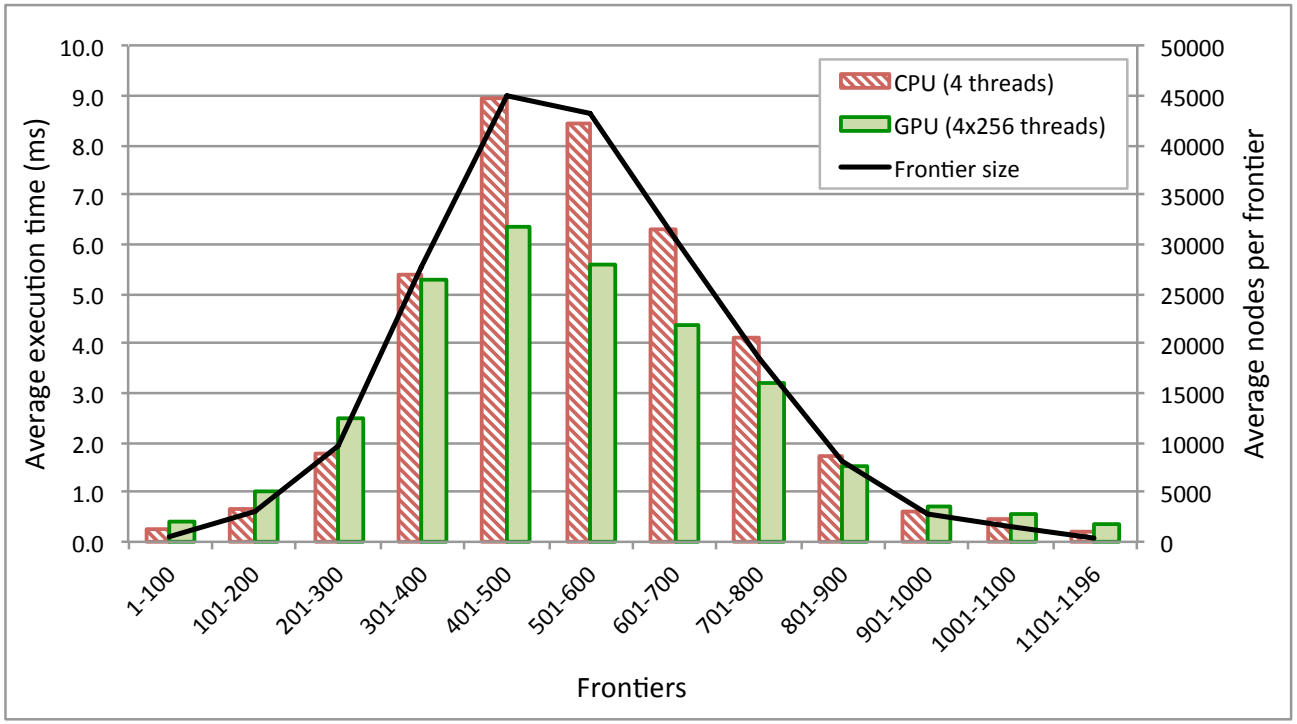


https://youtu.be/X74BLPO8tT4

Collaborative Computing

Recall: BFS on CPU or GPU?

- Motivation
  - Small-sized frontiers underutilize GPU resources
    - NVIDIA Jetson TX1 (4 ARMv8 CPU cores + 2 GPU cores)
    - New York City roads



BFS: Collaborative Implementation (I)

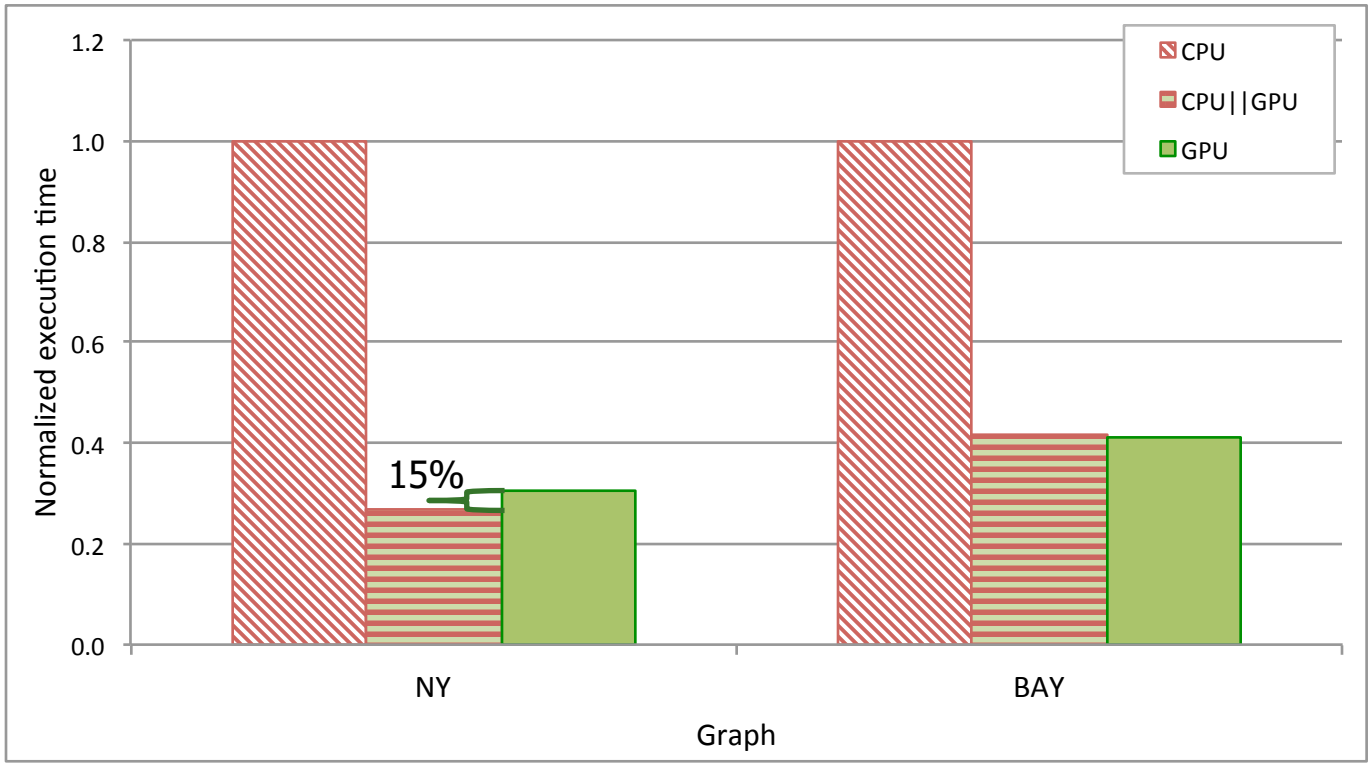
- Choose CPU or GPU depending on frontier

```
// Host code
while(frontier_size != 0){

    if(frontier_size < LIMIT){

        // Launch CPU threads
    }
    else{

        // Launch GPU kernel
    }
}
```

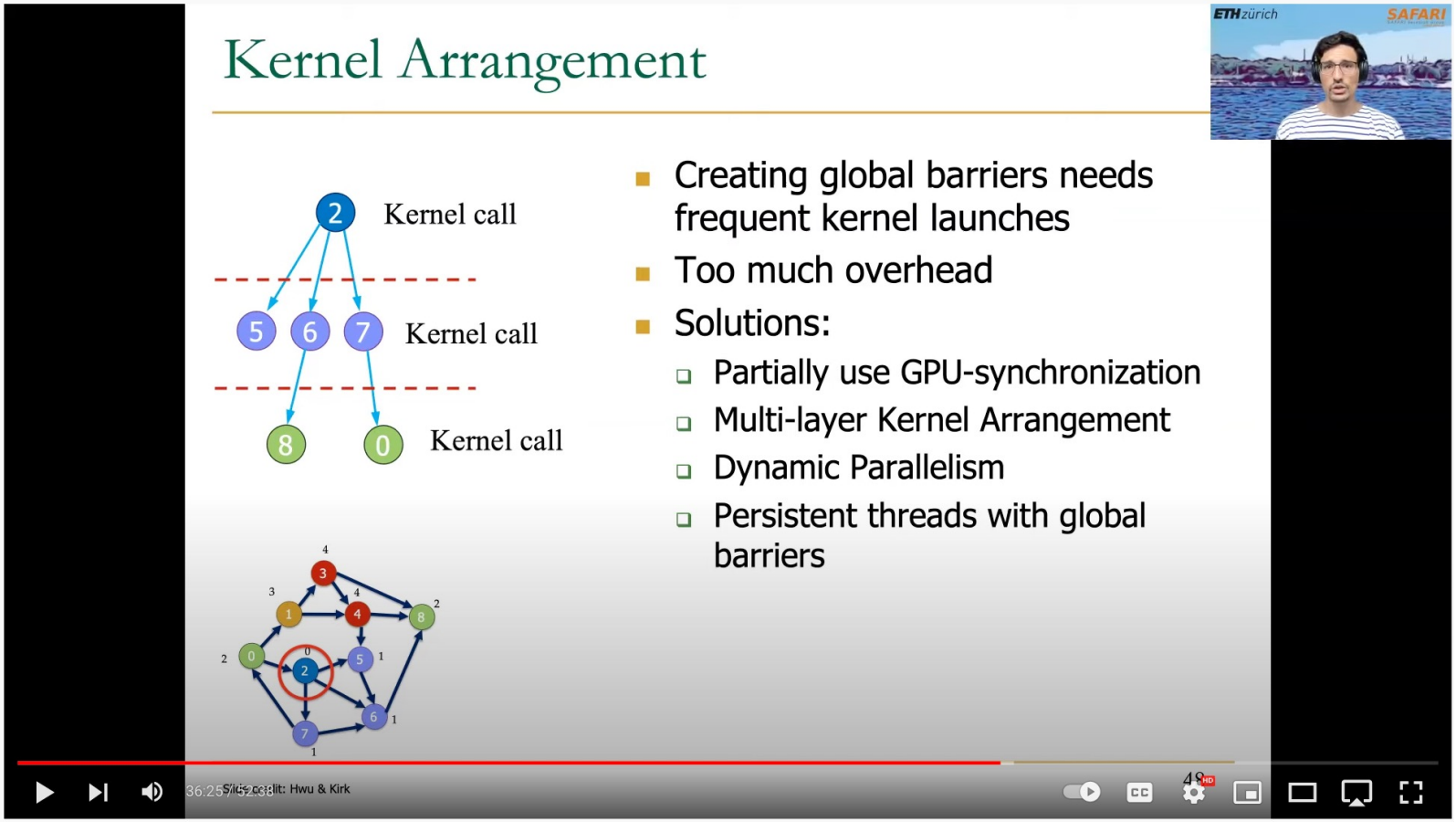
- CPU threads or GPU kernel keep running while the condition is satisfied

BFS: Collaborative Implementation (II)

- Experimental results
  - NVIDIA Jetson TX1 (4 ARMv8 CPU cores + 2 GPU cores)



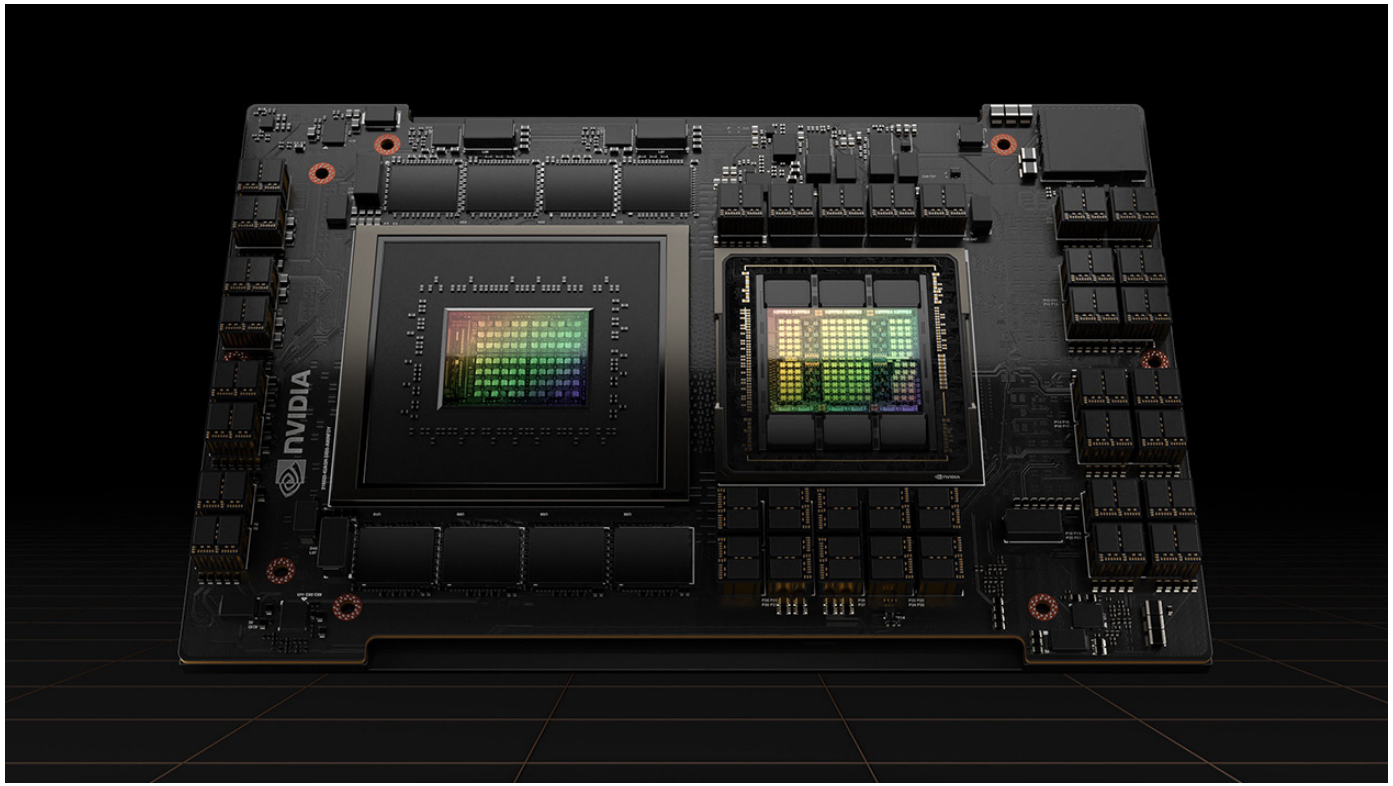
Lecture on Graph Search



https://youtu.be/cvP1aH31ar4

NVIDIA Grace Hopper Superchip

- CPU + GPU
  - Grace CPU + Hopper GPU
- 900 GB/s coherent interface (7x faster than PCIe Gen 5)



https://www.nvidia.com/en-us/data-center/grace-cpu/

Unified Memory

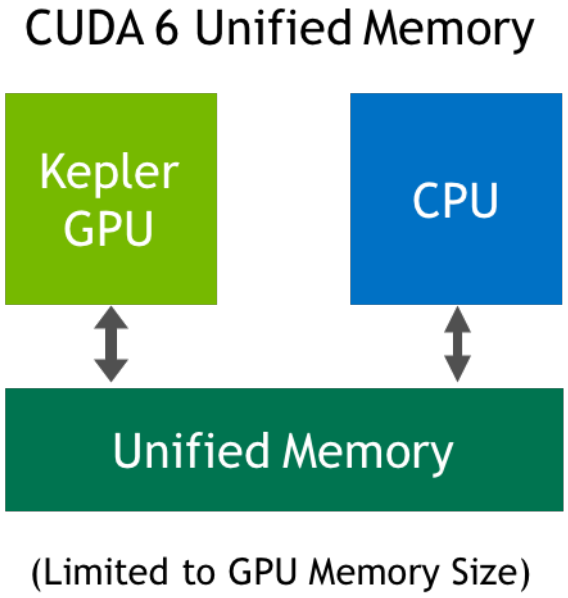
Memory Allocation and Data Transfers

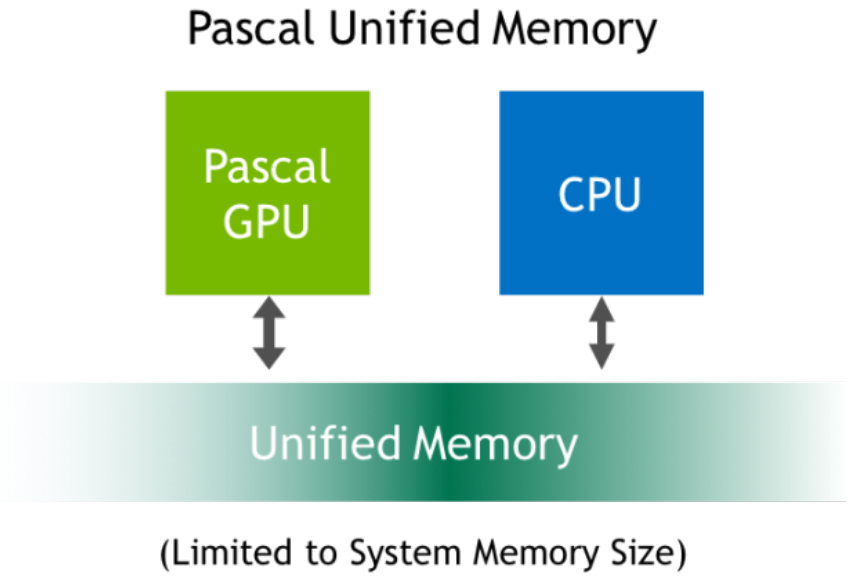
- Traditional approach to device allocation, CPU-GPU transfer, and GPU-CPU transfer
  - `cudaMalloc();`
  - `cudaMemcpy();`
- Naturally matches systems with discrete GPUs

```
// Allocate input
malloc(input, ...);
cudaMalloc(d_input, ...);
cudaMemcpy(d_input, input, ..., HostToDevice); // Copy to device memory

// Allocate output
malloc(output, ...);
cudaMalloc(d_output, ...);

// Launch GPU kernel
gpu_kernel<<<blocks, threads>>> (d_output, d_input, ...);

// Synchronize
cudaDeviceSynchronize();

// Copy output to host memory
cudaMemcpy(output, d_output, ..., DeviceToHost);
```

Unified Memory

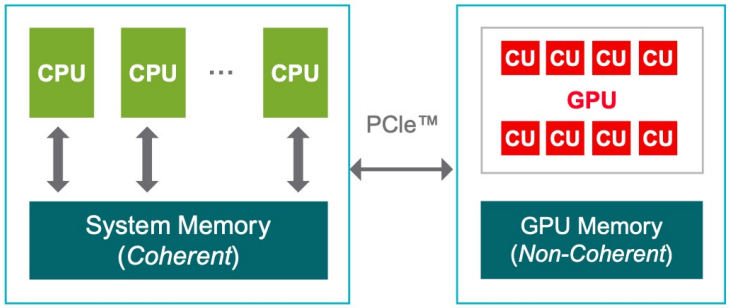
- Unified Virtual Address space
  - Same virtual address space across host and device
- CUDA 6.0: Unified memory
- CUDA 8.0 + Pascal: GPU page faults



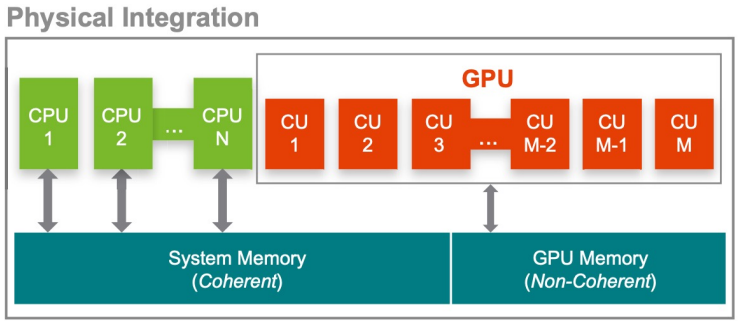


Heterogeneous System Architecture

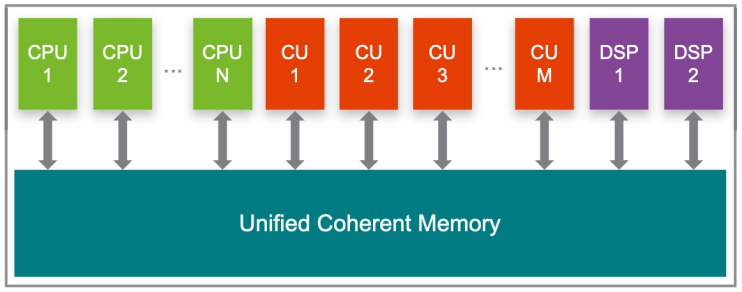
- HSA extends the unified memory space beyond GPUs
  - DSPs, DMA engines, cryptoengines, and other accelerators



Legacy GPU compute on discrete GPU cards



Legacy GPU compute on SOCs



An HSA enabled SOC featuring multiple processors beyond CPU

Hwu (editor), "Heterogeneous System Architecture: A New Compute Platform Infrastructure," 2016

Unified Memory: Memory Management

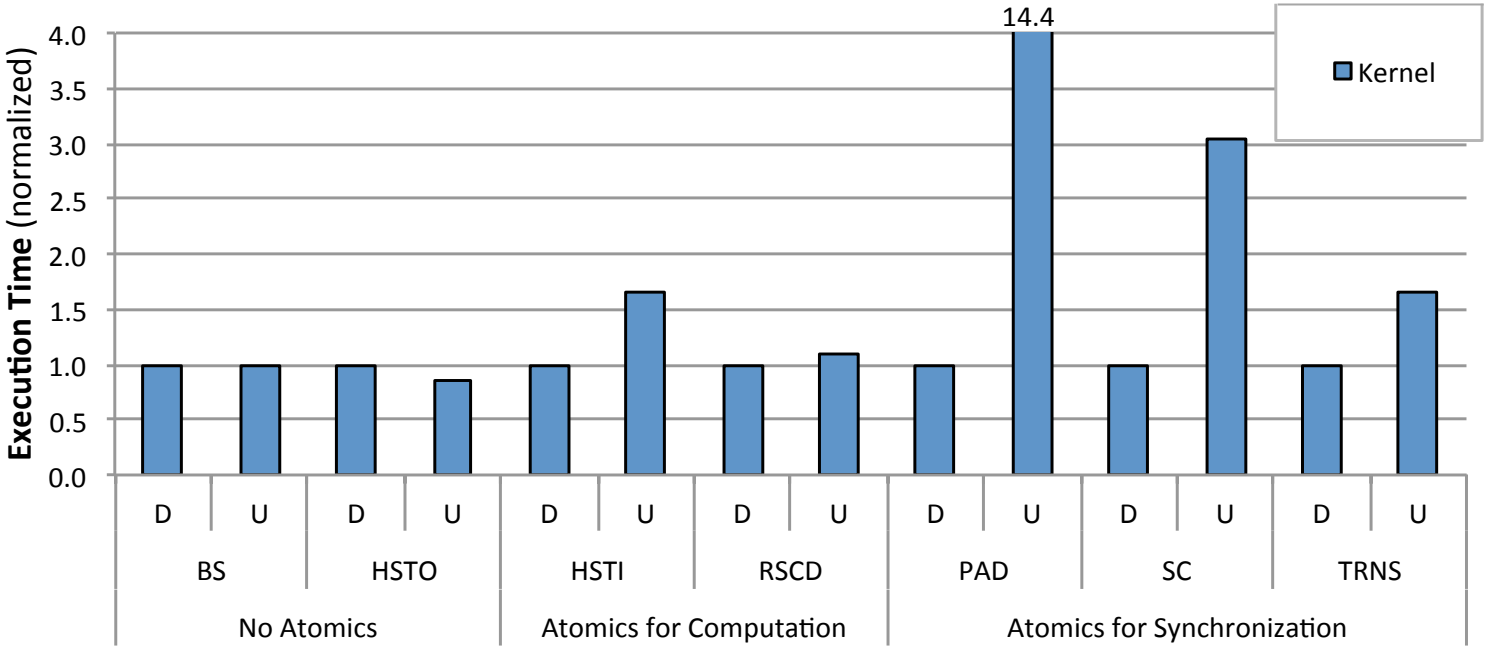
- Easier programming with Unified Memory
  - `cudaMallocManaged();`

```
// Allocate input
malloc(input, ...);
cudaMallocManaged(d_input, ...);
memcpy(d_input, input, ...); // Copy to managed memory

// Allocate output
cudaMallocManaged(d_output, ...);

// Launch GPU kernel
gpu_kernel<<<blocks, threads>>> (d_output, d_input, ...);

// Synchronize
cudaDeviceSynchronize();
```

- No need for double allocation or explicit data transfers
- Naturally matches physically integrated devices (e.g., CPU and GPU in the same chip) or devices with the same physical memory (e.g., CPU and GPU in the same package)
  - But it can also be implemented for discrete GPUs

Unified Memory: Kernel Time

- IBM Power8 with NVIDIA Pascal GPU
  - D: Discrete (or traditional, without unified memory)
  - U: Unified memory

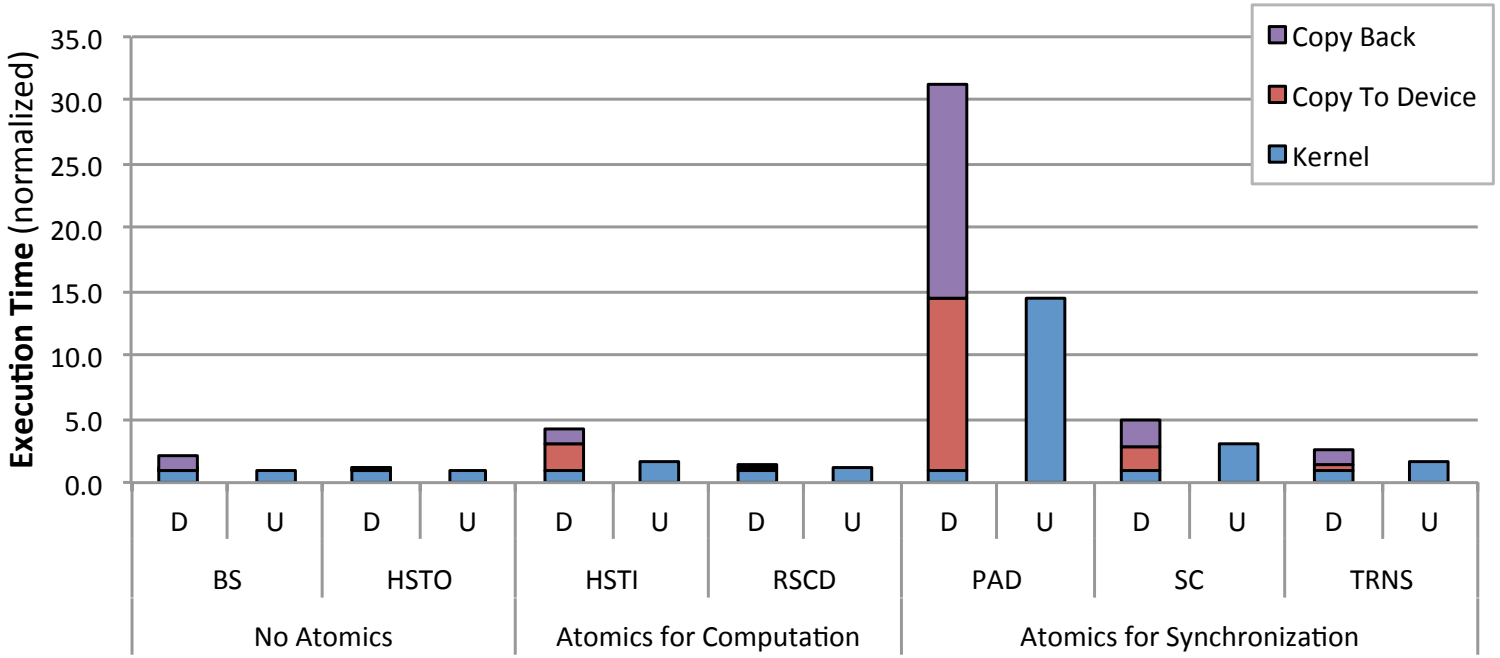


No cross-device communication

Cross-device communication may heavily burden kernel performance

Unified Memory: Total Execution Time

- IBM Power8 with NVIDIA Pascal GPU
  - D: Discrete (or traditional, without unified memory)
  - U: Unified memory



Unified memory can hide data transfers with kernel execution

How to Implement Collaborative Computing Applications?

Collaborative Computing Applications

- Case studies using CPU and GPU
- Kernel launches are asynchronous
  - CPU can work while waits for GPU to finish
  - Traditionally, this is the most efficient way to exploit heterogeneity

```
// Allocate input
malloc(input, ...);
cudaMalloc(d_input, ...);
cudaMemcpy(d_input, input, ..., HostToDevice); // Copy to device memory

// Allocate output
malloc(output, ...);
cudaMalloc(d_output, ...);

// Launch GPU kernel
gpu_kernel<<<blocks, threads>>> (d_output, d_input, ...);

// CPU can do things here

// Synchronize
cudaDeviceSynchronize();

// Copy output to host memory
cudaMemcpy(output, d_output, ..., DeviceToHost);
```

Fine-Grained Collaboration

- Fine-grained collaboration becomes possible with unified memory (post Kepler/Maxwell architecture)
- Pascal/Volta/Turing/Ampere Unified Memory (& HSA)
  - CPU-GPU memory coherence
  - System-wide atomic operations

```
// Allocate input
cudaMallocManaged(input, ...);

// Allocate output
cudaMallocManaged(output, ...);

// Launch GPU kernel
gpu_kernel<<<blocks, threads>>> (output, input, ...);

// CPU can do things here
output[x] = input[y];

output[x+1].fetch_add(1);
```

CUDA 8.0 and Later

- Unified memory

```
cudaMallocManaged(&h_in, in_size);
```

- System-wide atomics

```
old = atomicAdd_system(&h_out[x], inc);
```

OpenCL 2.0 and Later

- Shared virtual memory

```
XYZ * h_in = (XYZ *)clSVMAlloc(
            ocl.clContext, CL_MEM_SVM_FINE_GRAIN_BUFFER, in_size, 0);
```

- More flags:

```
CL_MEM_READ_WRITE
CL_MEM_SVM_ATOMICS
```

- C++11 atomic operations

(`memory_scope_all_svm_devices`)

```
old = atomic_fetch_add(&h_out[x], inc);
```

C++AMP (HCC)

- Unified memory space (HSA)

```
XYZ *h_in = (XYZ *)malloc(in_size);
```

- C++11 atomic operations

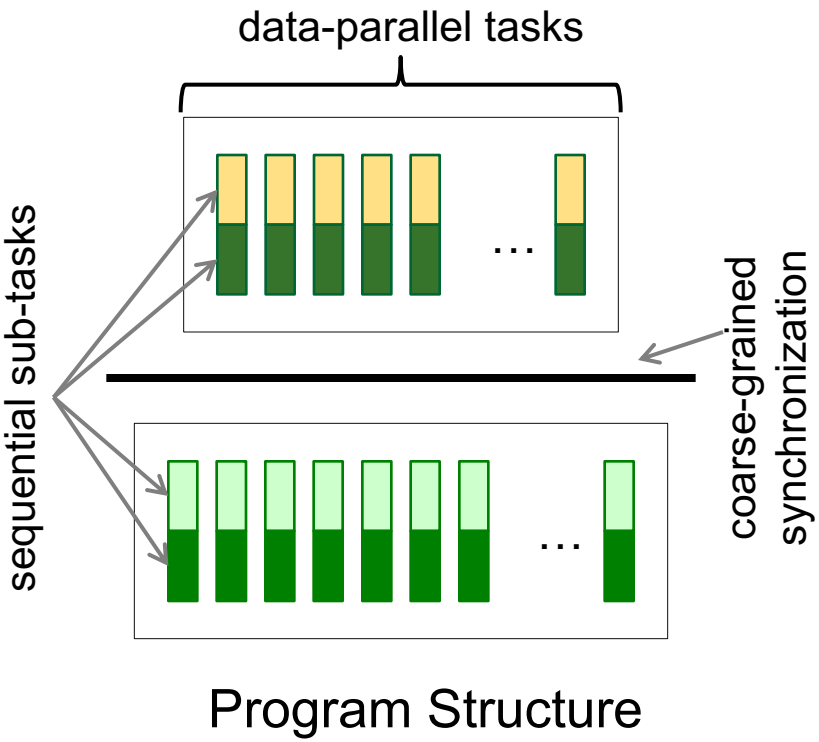
  (`memory_scope_all_svm_devices`)

  - Platform atomics (HSA)

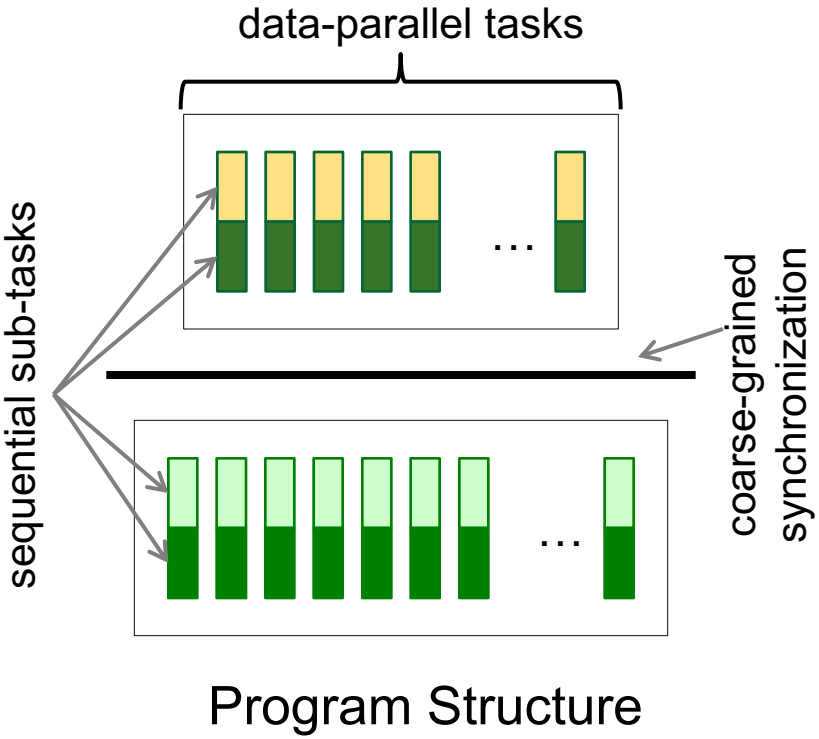
```
old = atomic_fetch_add(&h_out[x], inc);
```

Collaborative Patterns

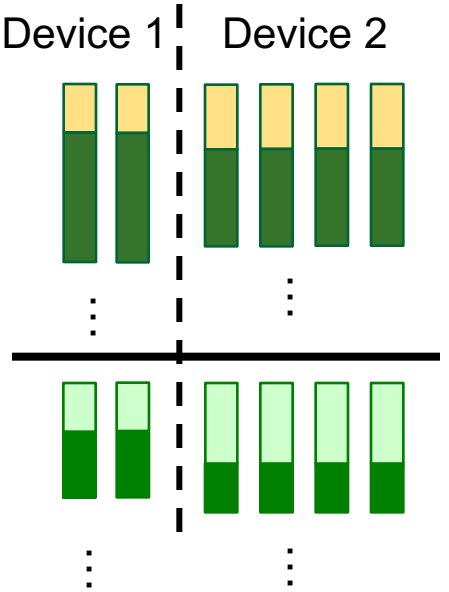
Traditional Program Structure



Collaborative Patterns: Data Partitioning

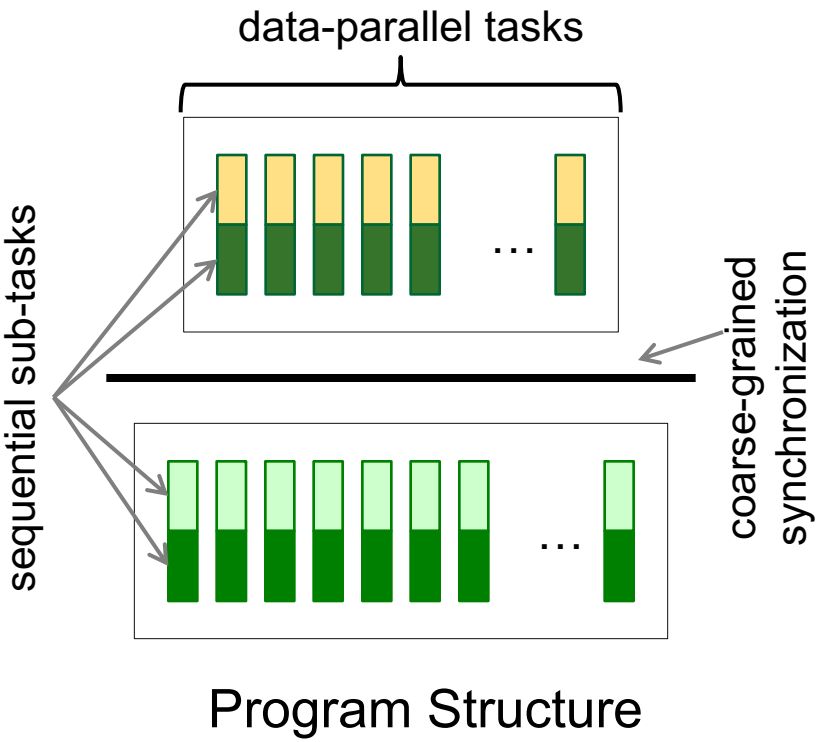


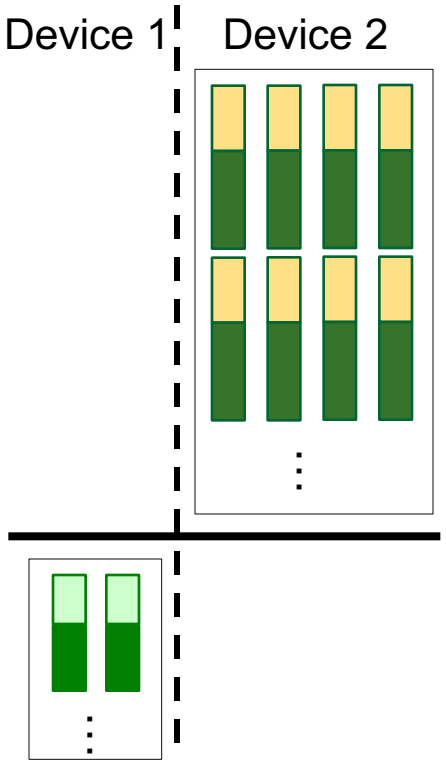
Program Structure



Data Partitioning

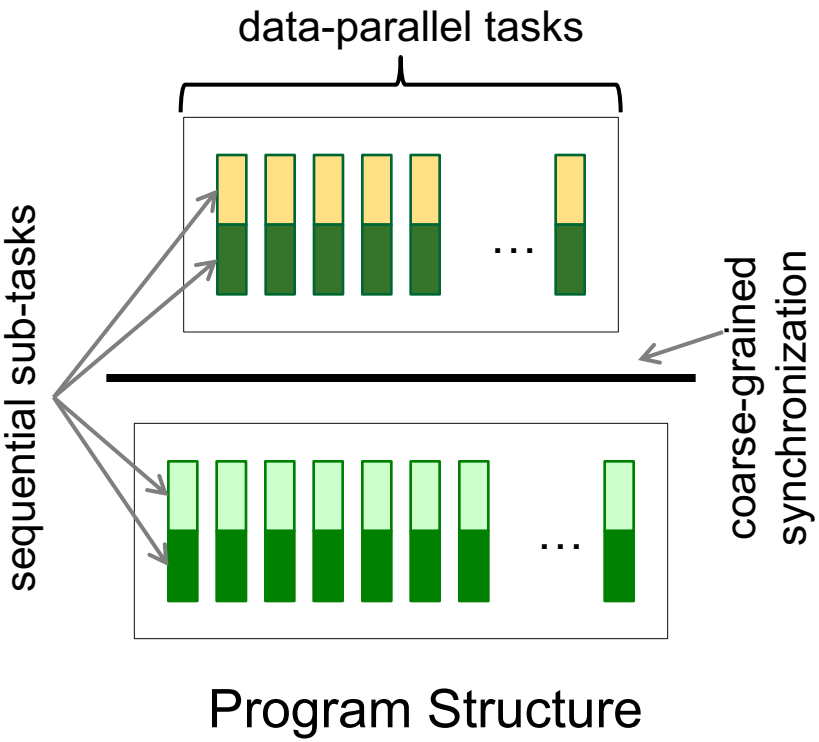
Collaborative Patterns: Task Partitioning (I)

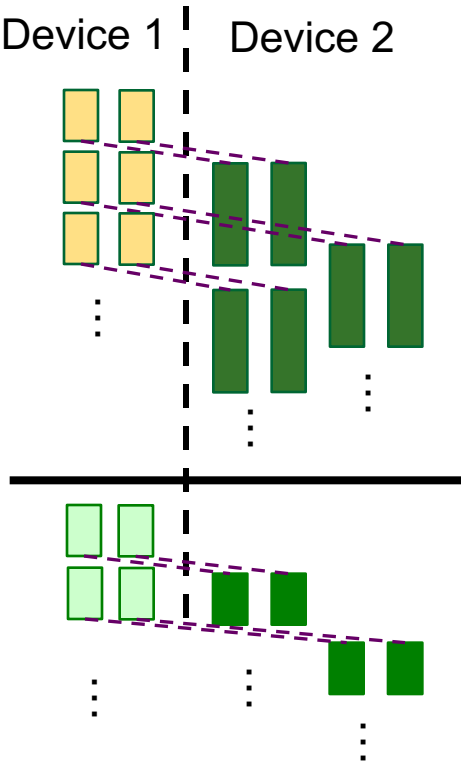




Coarse-grained Task Partitioning

Collaborative Patterns: Task Partitioning (II)



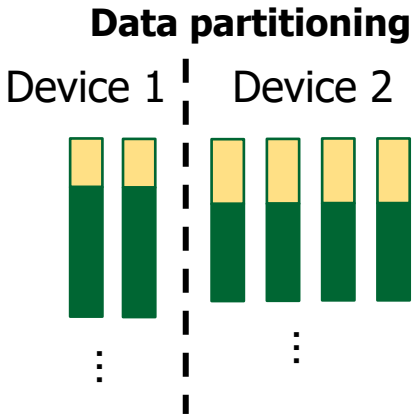


Analytical Modeling

- N: Number of data parallel tasks in the application
- $t_{i,D1}$: Execution time of sub-task i by a Device 1 worker
- $t_{i,D2}$: Execution time of sub-task i by a Device 2 worker
- w_{D1}: Number of available Device 1 workers
- w_{D2}: Number of available Device 2 workers
- β: Distribution and aggregation overhead factor
- α: Fraction of data parallel tasks assigned to Device 1
- S_{D1} and S_{D2} are, respectively, the set of subtasks/tasks executed in Device 1 and Device 2

Analytical Model: Data Partitioning

- N: Number of data parallel tasks in the application
- $t_{i,D1}$: Execution time of sub-task i by a Device 1 worker
- $t_{i,D2}$: Execution time of sub-task i by a Device 2 worker
- w_{D1}: Number of available Device 1 workers
- w_{D2}: Number of available Device 2 workers
- β: Distribution and aggregation overhead factor
- α: Fraction of data parallel tasks assigned to Device 1



Data partitioning

The total execution time is

Total D1 execution time (sequential execution)

Total D2 execution time (sequential execution)

$$t_{\text{data, total}} = \beta_{\text{data}} \cdot \max\left(\frac{\alpha N \sum_i t_{i,D1}}{w_{D1}}, \frac{(1-\alpha) N \sum_i t_{i,D2}}{w_{D2}}\right)$$

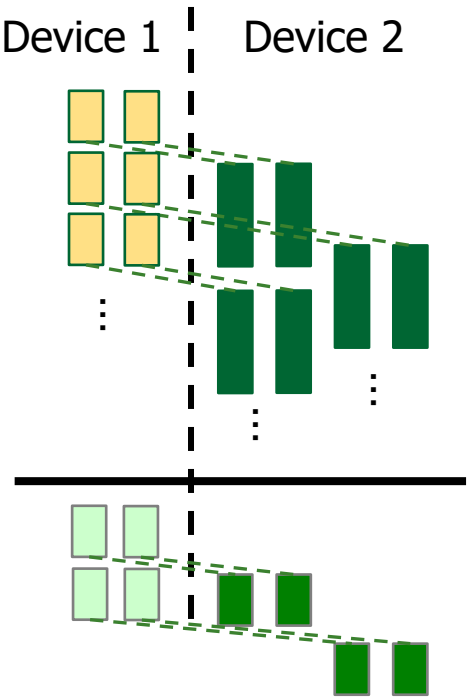
Fixing all the variables except α, the optimal α (global minimum point) is

$$\alpha^* = \frac{\sum_i t_{i,D2}}{w_{D2}} \Big/ \left(\frac{\sum_i t_{i,D1}}{w_{D1}} + \frac{\sum_i t_{i,D2}}{w_{D2}}\right)$$

Workloads of Device 1 and Device 2 workers are balanced

Analytical Model: Fine-Grained Task Part.

- N: Number of data parallel tasks in the application
- $t_{i,D1}$: Execution time of sub-task i by a Device 1 worker
- $t_{i,D2}$: Execution time of sub-task i by a Device 2 worker
- w_{D1}: Number of available Device 1 workers
- w_{D2}: Number of available Device 2 workers
- β: Distribution and aggregation overhead factor
- S_{D1} and S_{D2} are, respectively, the set of subtasks executed in Device 1 and Device 2



Fine-grained task partitioning

The total execution time is

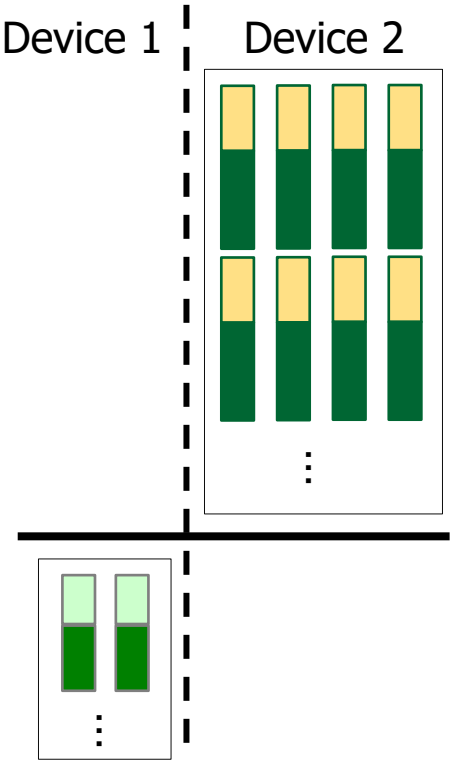
$$t_{\text{task, total}} = \beta_{\text{task}} N \cdot \max\left(\frac{\sum_{i \in S_{D1}} t_{i,D1}}{w_{D1}}, \frac{\sum_{i \in S_{D2}} t_{i,D2}}{w_{D2}}\right)$$

(Assume sub-tasks are very fine-grained)

Analytical Model: Coarse-Grained Task Part.

- N: Number of data parallel tasks in the application
- $t_{i,D1}$: Execution time of sub-task i by a Device 1 worker
- $t_{i,D2}$: Execution time of sub-task i by a Device 2 worker
- w_{D1}: Number of available Device 1 workers
- w_{D2}: Number of available Device 2 workers
- β: Distribution and aggregation overhead factor
- S_{D1} and S_{D2} are, respectively, the set of tasks executed in Device 1 and Device 2

Coarse-grained task partitioning



Coarse-grained task partitioning

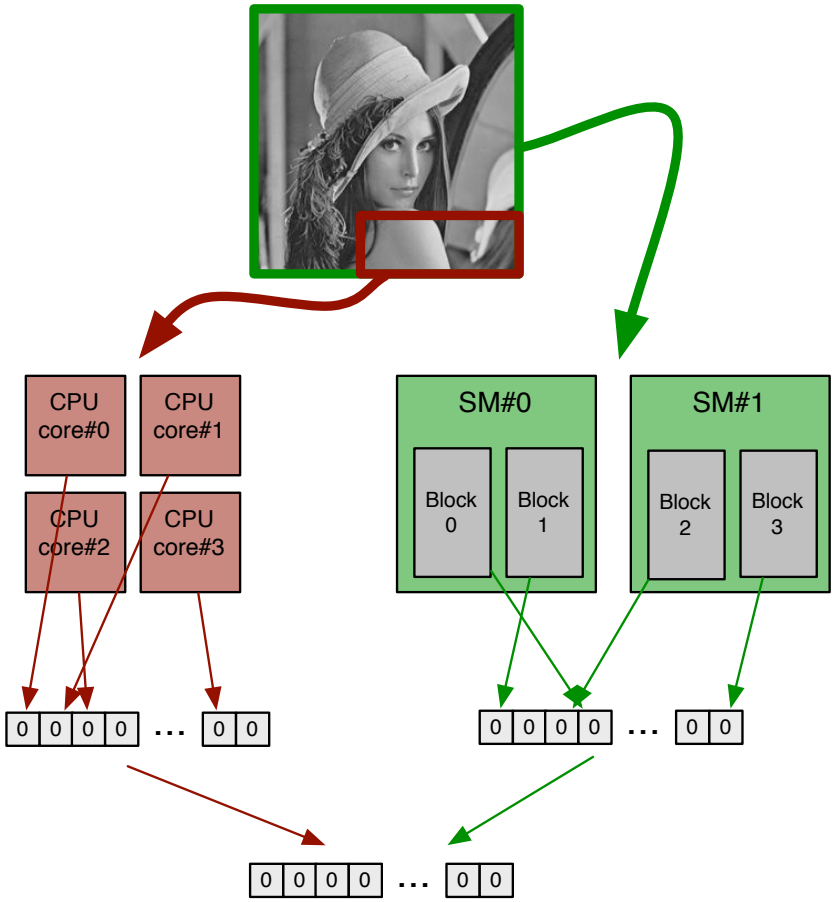
The total execution time is

$$t_{\text{task, total}} = \beta_{\text{task}} N \cdot \left(\frac{\sum_{i \in S_{D1}} t_{i,D1}}{w_{D1}} + \frac{\sum_{i \in S_{D2}} t_{i,D2}}{w_{D2}} \right)$$

Data Partitioning

Histogram without Unified Memory

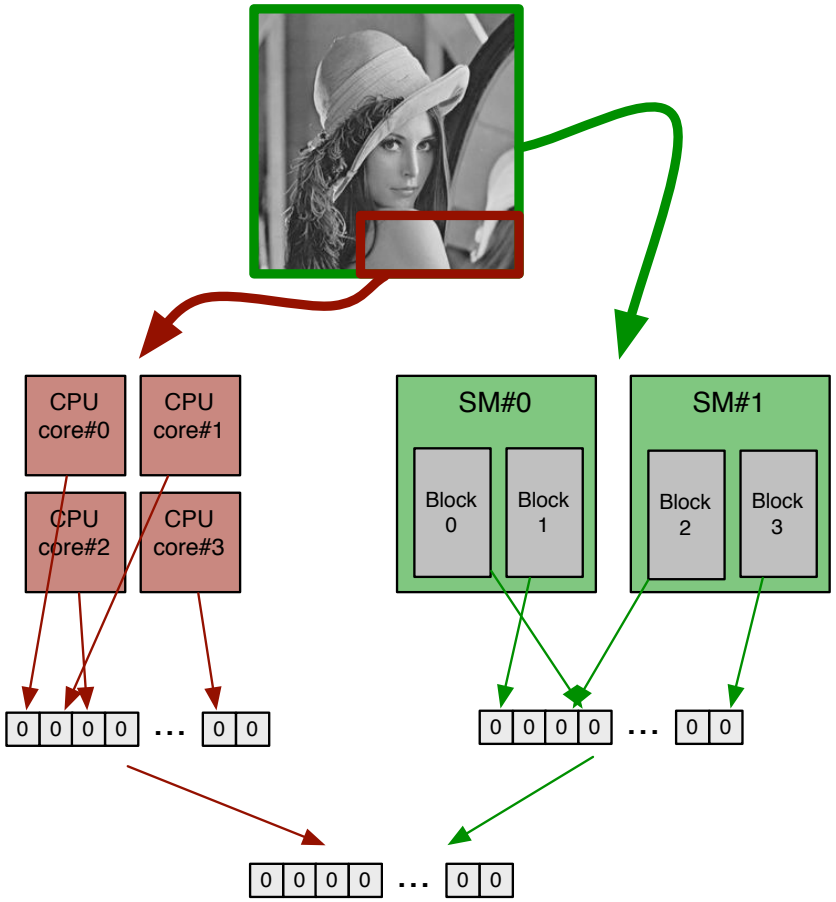
- Traditional approach: Separate CPU and GPU histograms are merged at the end




```
malloc(CPU image);
cudaMalloc(GPU image);
cudaMemcpy(GPU image, CPU image, ...,
           HosttoDevice);
malloc(CPU histogram);
memset(CPU histogram, 0);
cudaMalloc(GPU histogram);
cudaMemset(GPU histogram, 0);

// Launch CPU threads
// Launch GPU kernel

cudaMemcpy(GPU histogram, DeviceToHost);

// Launch CPU threads for merging
```

Histogram with Unified Memory (I)

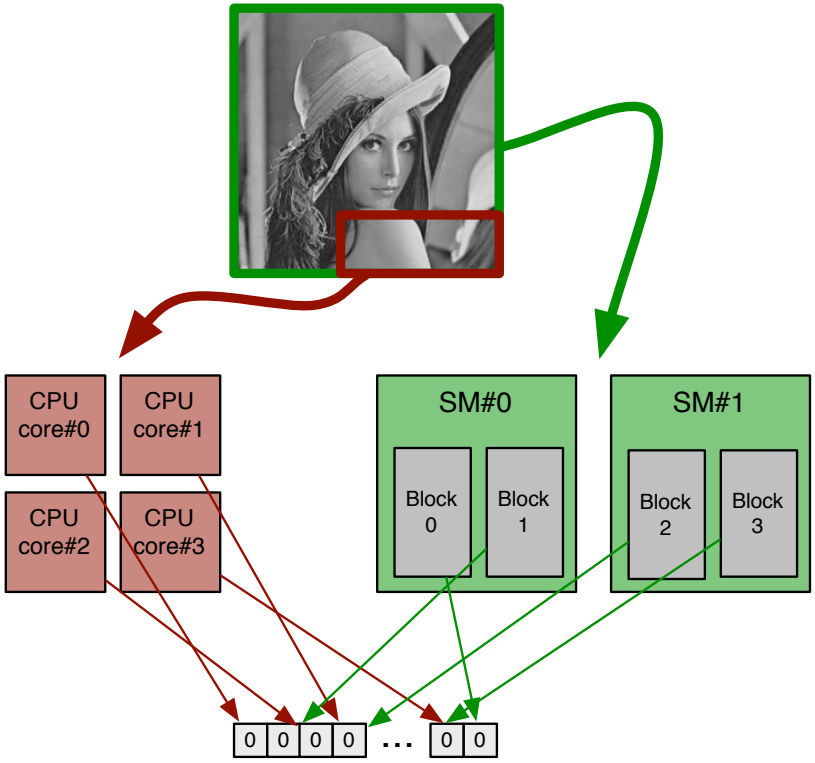
- Traditional approach: Separate CPU and GPU histograms are merged at the end




```
malloc(CPU image);
cudaMallocManaged(GPU image);
memcpy(GPU image, CPU image, ...);

malloc(CPU histogram);
memset(CPU histogram, 0);
cudaMallocManaged(GPU histogram);
cudaMemset(GPU histogram, 0);

// Launch CPU threads
// Launch GPU kernel

cudaDeviceSynchronize();

// Launch CPU threads for merging
```

Histogram with Unified Memory (II)

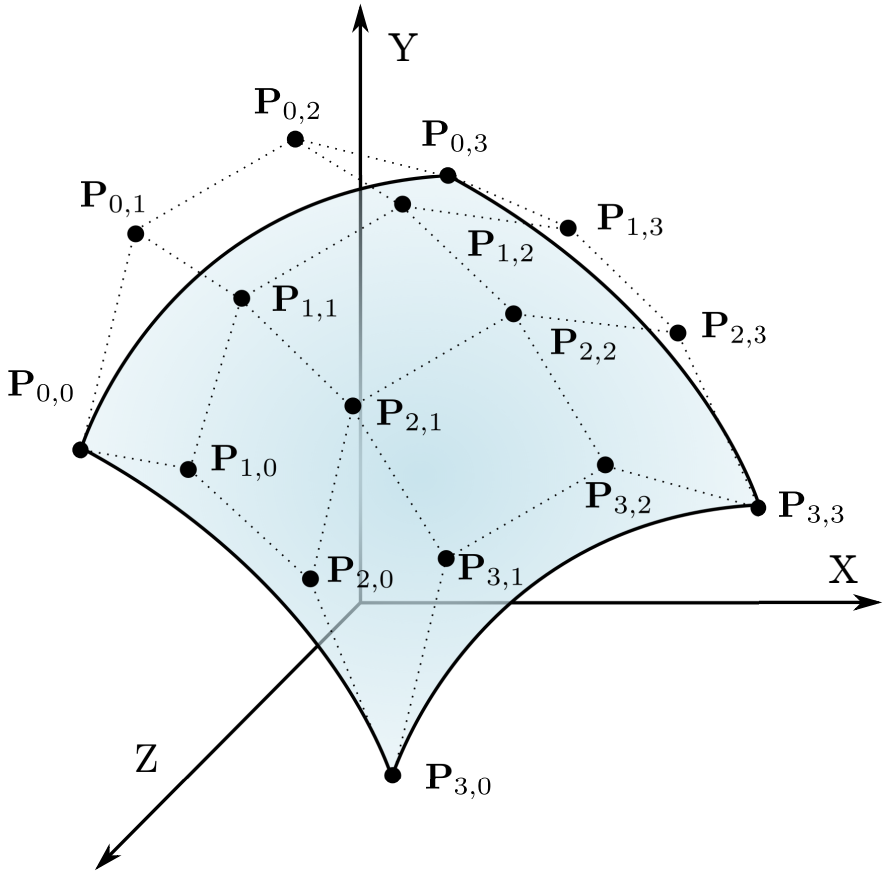
- System-wide atomic operations: One single histogram




```
malloc(CPU image);
cudaMallocManaged(GPU image);
memcpy(GPU image, CPU image, ...);

cudaMallocManaged(Histogram);
cudaMemset(Histogram, 0);

// Launch CPU threads
// Launch GPU kernel (atomicAdd_system)
```

Bézier Surfaces (I)

- Bézier surface: 4x4 net of control points



Bézier Surfaces (II)

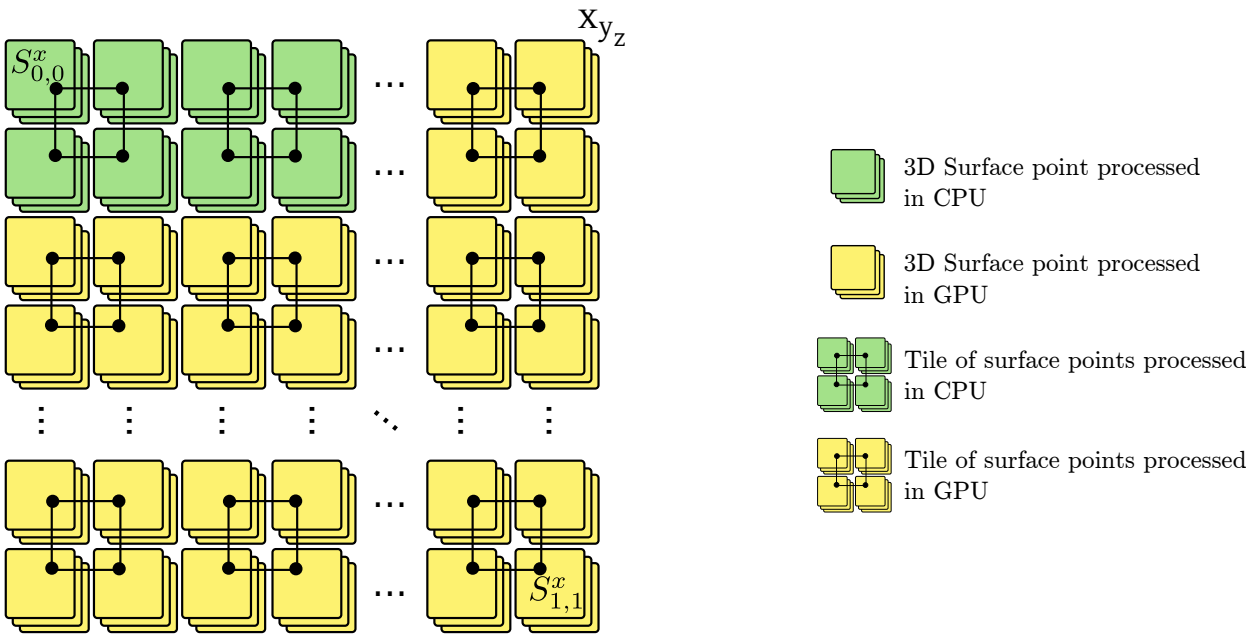
- Parametric non-rational formulation
  - Bernstein polynomials
  - Bi-cubic surface $m = n = 3$

$$\mathbf{S}(u,v) = \sum_{i=0}^{m}\sum_{j=0}^{n}\mathbf{P}_{i,j}B_{i,m}(u)B_{j,n}(v), \qquad (1)$$

$$B_{i,m}(u) = \binom{m}{i}(1-u)^{(m-i)}u^{i}, \qquad (2)$$

Bézier Surfaces: Static Distribution (I)

- Collaborative implementation
  - Tiles calculated by GPU blocks or CPU threads
  - Static distribution



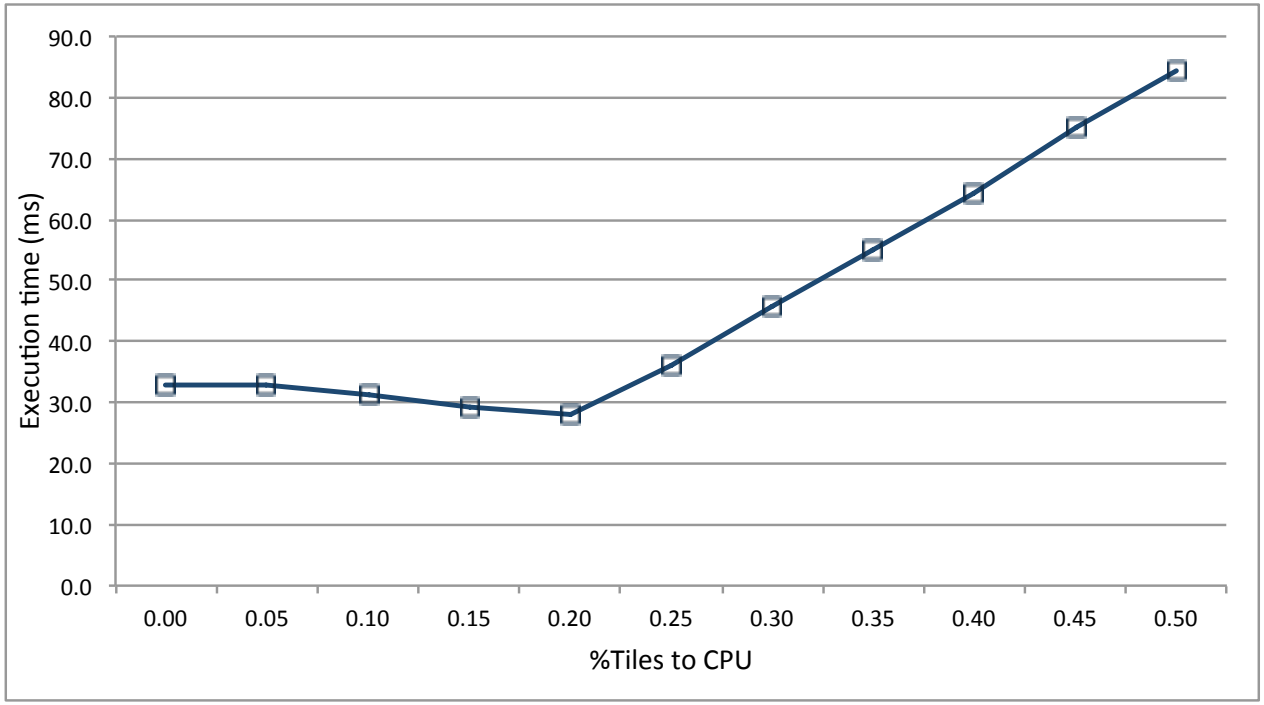
Bézier Surfaces: Static Distribution (II)

- Without Unified Memory

```
// Allocate control points
malloc(control_points, ...);
generate_cp(control_points);
cudaMalloc(d_control_points, ...);
cudaMemcpy(d_control_points, control_points, ..., HostToDevice); // Copy to device memory

// Allocate surface
malloc(surface, ...);
cudaMalloc(d_surface, ...);

// Launch CPU threads
std::thread main_thread (run_cpu_threads, control_points, surface, ...);

// Launch GPU kernel
gpu_kernel<<<blocks, threads>>> (d_surface, d_control_points, ...);

// Synchronize
main_thread.join();
cudaDeviceSynchronize();

// Copy GPU part of surface to host memory
cudaMemcpy(&surface[end_of_cpu_part], d_surface, ..., DeviceToHost);
```

Bézier Surfaces: Static Distribution (III)

- Performance results on NVIDIA Jetson TX1 (4 ARMv8 CPU cores + 2 GPU cores)
  - Bezier surface: 300x300, 4x4 control points
  - %Tiles to CPU
  - 17% speedup over GPU only



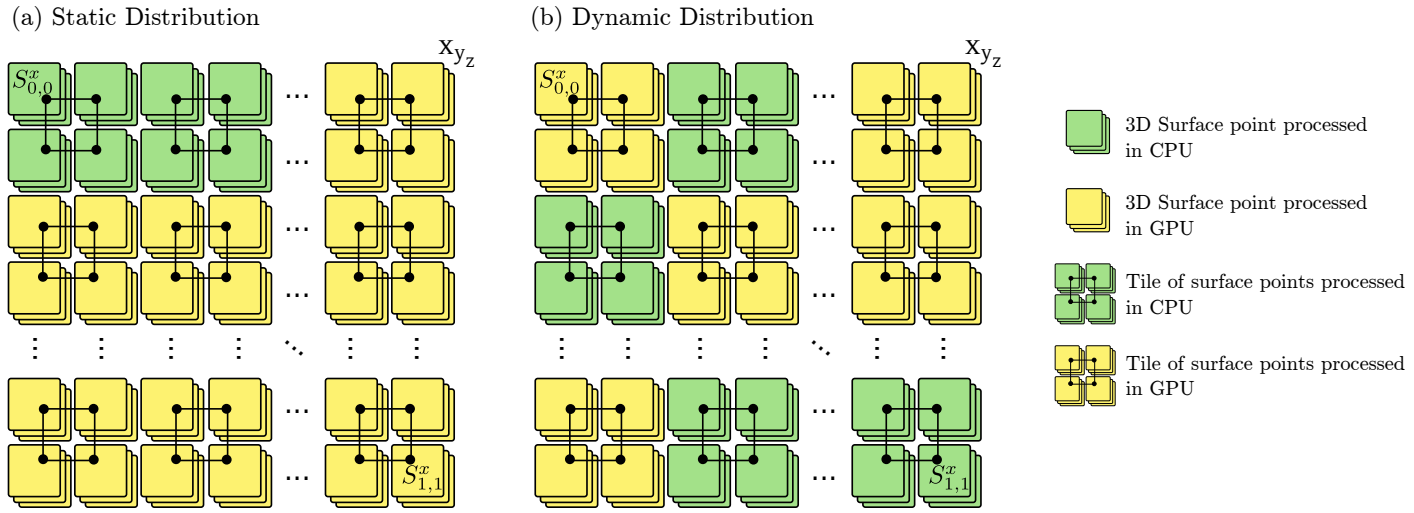
Bézier Surfaces with Unified Memory

- With Unified Memory

```
// Allocate control points
malloc(control_points, ...);
generate_cp(control_points);
cudaMalloc(d_control_points, ...);
cudaMemcpy(d_control_points, control_points, ..., HostToDevice); // Copy to device memory

// Allocate surface
cudaMallocManaged(surface, ...);

// Launch CPU threads
std::thread main_thread (run_cpu_threads, control_points, surface, ...);

// Launch GPU kernel
gpu_kernel<<<blocks, threads>>> (surface, d_control_points, ...);

// Synchronize
main_thread.join();
cudaDeviceSynchronize();
```

Bézier Surfaces: Dynamic Distribution

- Static vs. dynamic implementation

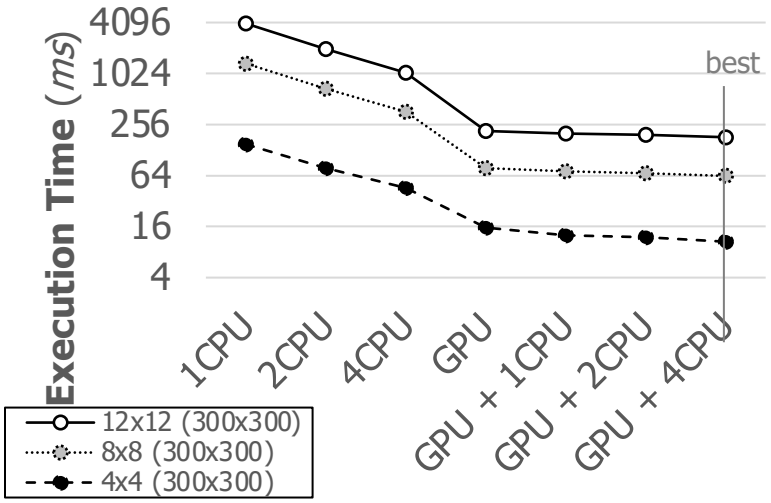


  - Pascal/Volta/Turing/Ampere Unified Memory: system-wide atomic operations

```
while(true){
    if(threadIdx.x == 0)
        my_tile = atomicAdd_system(tile_num, 1);  // my_tile in shared memory; tile_num in UM

    __syncthreads();  // Synchronization

    if(my_tile >= number_of_tiles) break;  // Break when all tiles processed

    ... // Kernel body
}
```

Benefits of Collaboration: Bézier Surfaces

- AMD Kaveri (4 CPU cores + 8 GPU cores)
  - Data partitioning improves performance

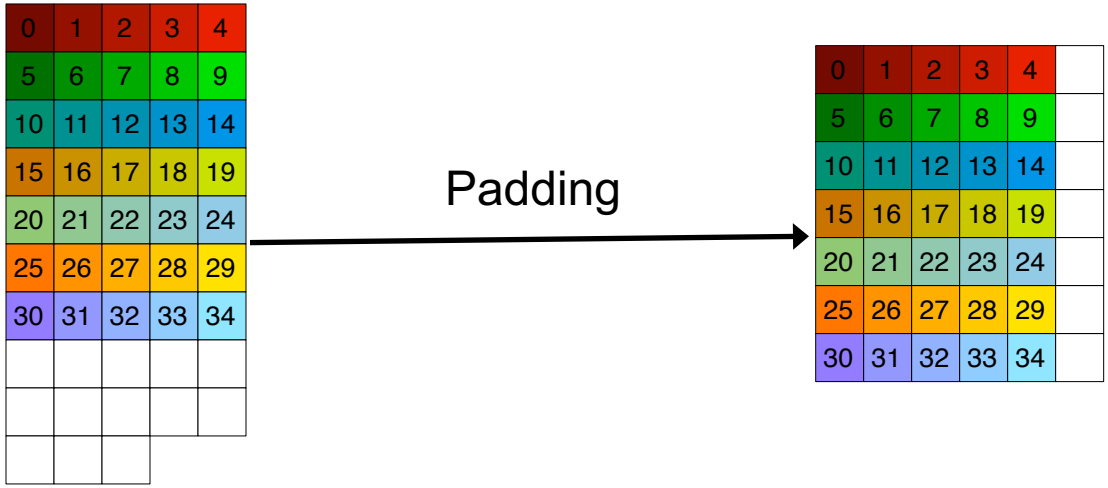


Bézier Surfaces

(up to 47% improvement over GPU only)

Padding (I)

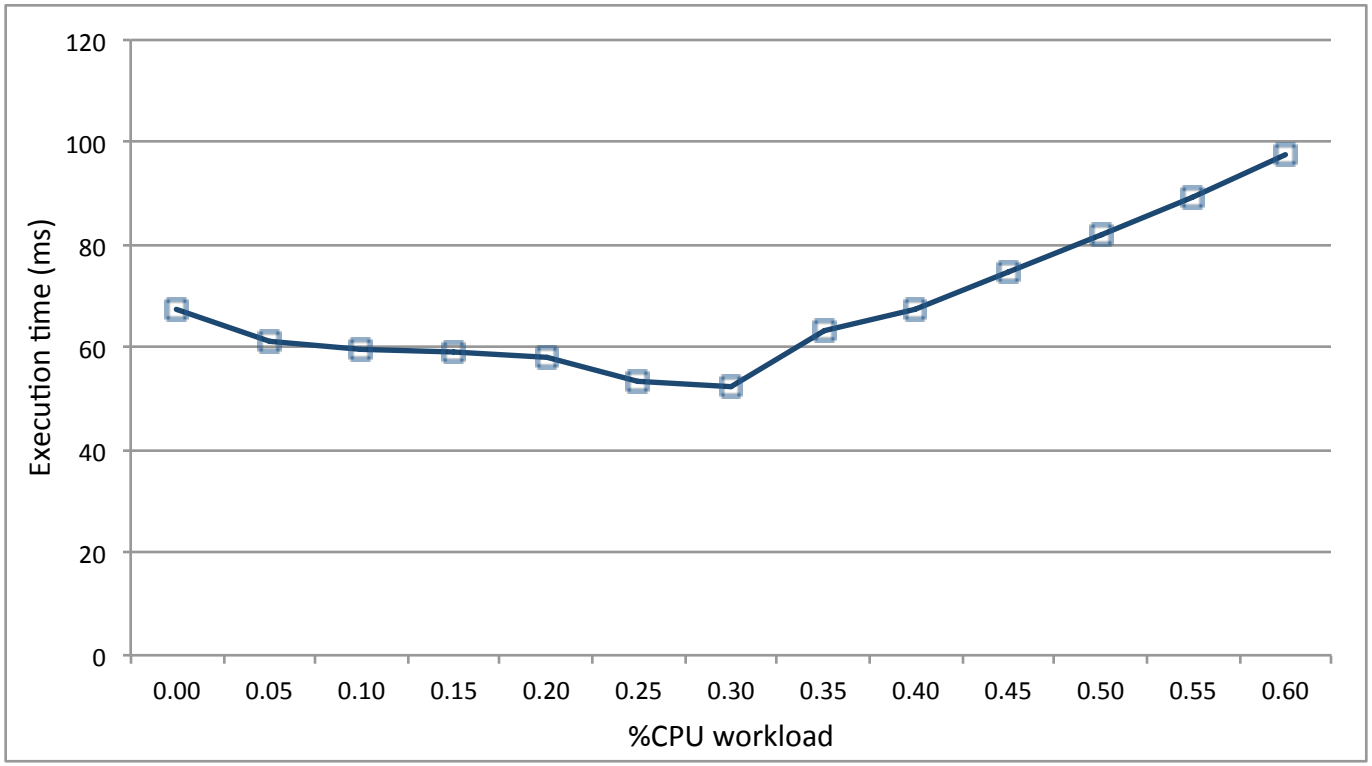
- Matrix padding
  - Use cases:
    - Memory alignment
    - Transposition of near-square matrices
    - Etc.



- Traditionally, it can only be performed out-of-place

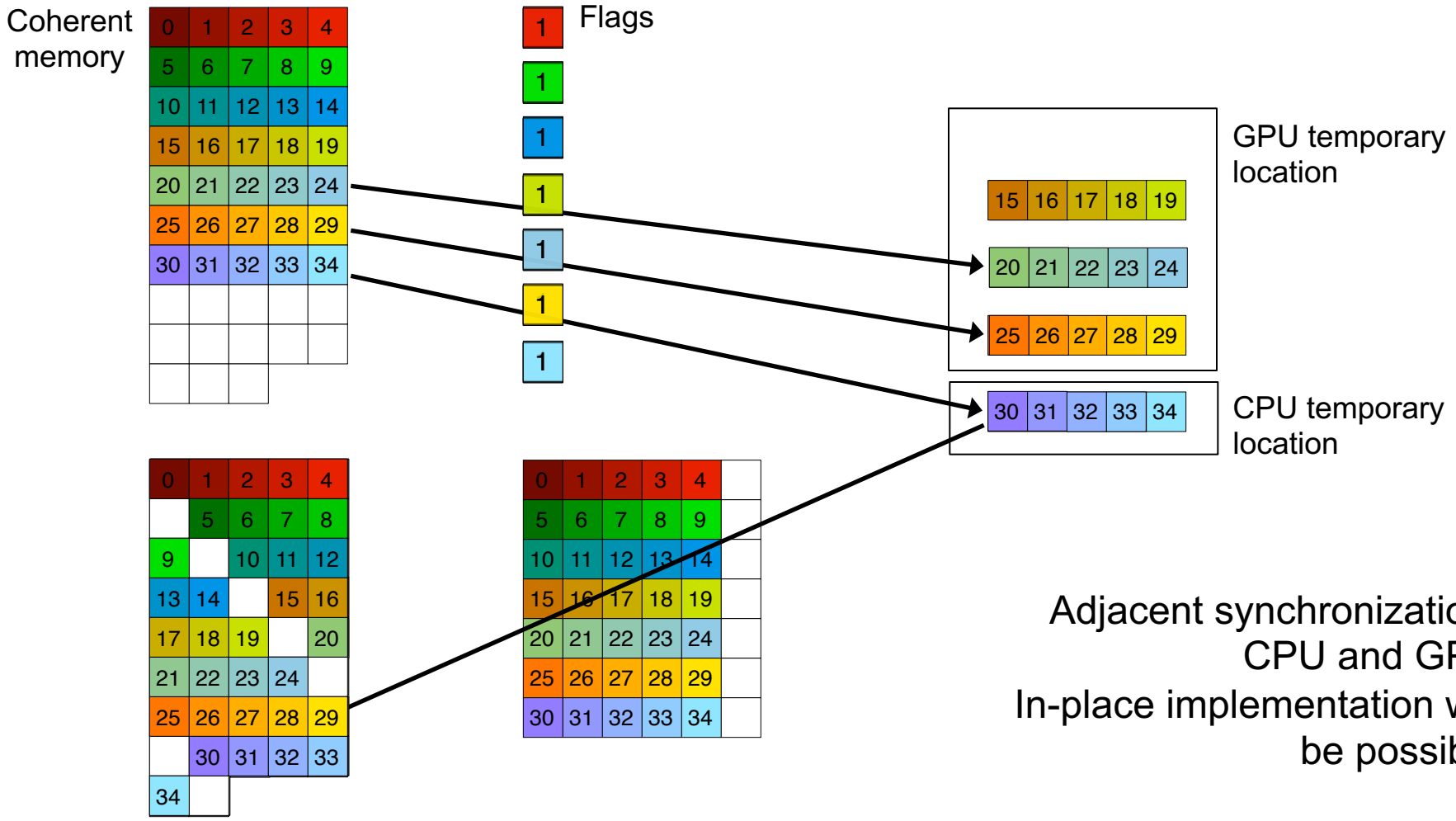
Padding (II)

- Performance results on NVIDIA Jetson TX1 (4 ARMv8 CPU cores + 2 GPU cores)
  - Matrix size: 4000x4000, padding = 1
  - 29% speedup over GPU only



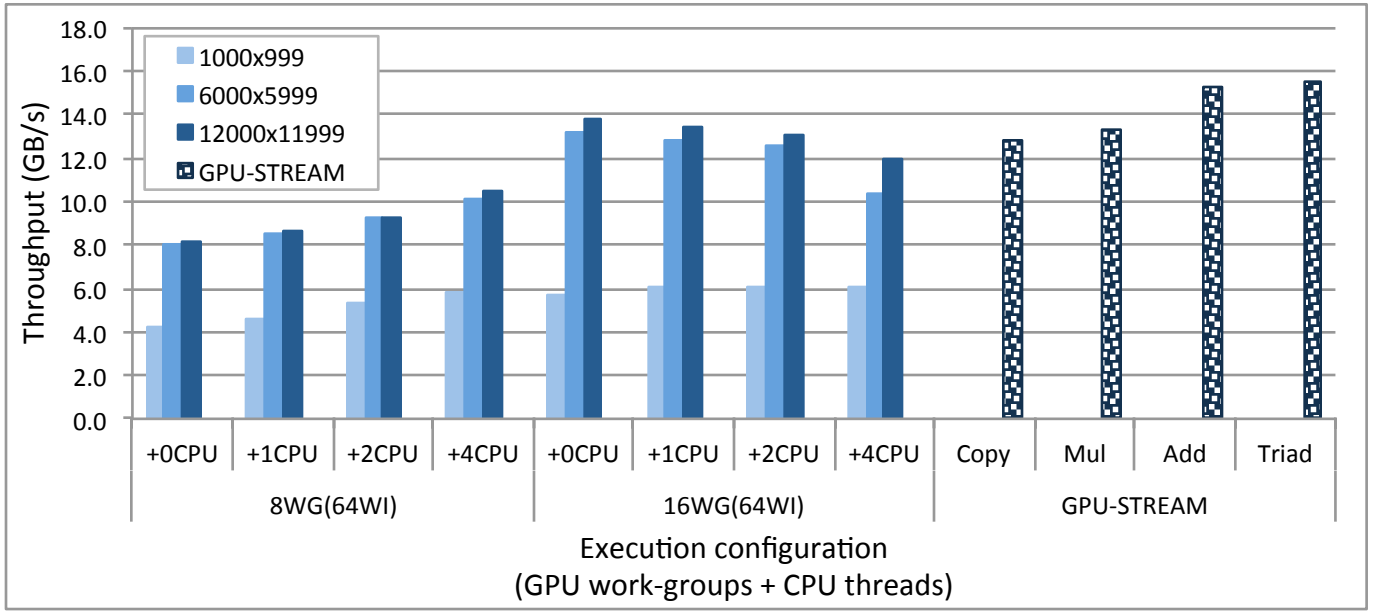
In-Place Padding

- Pascal/Volta/Turing/Ampere Unified Memory



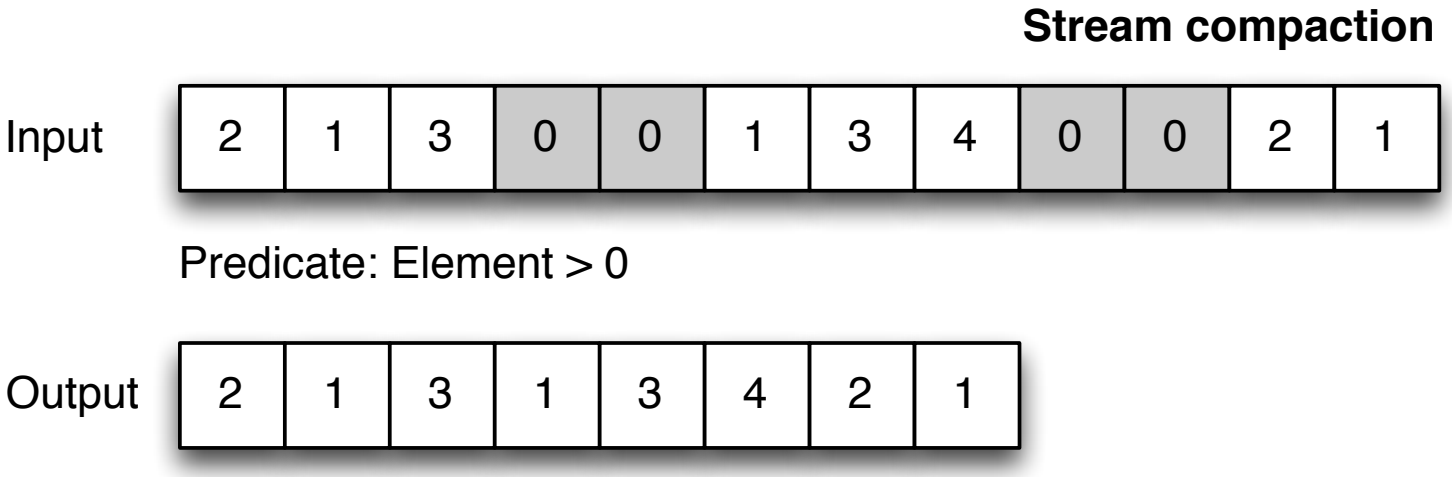
Benefits of Collaboration: Padding

- AMD Kaveri (4 CPU cores + 8 GPU cores)
  - Optimal number of devices is not always the maximum



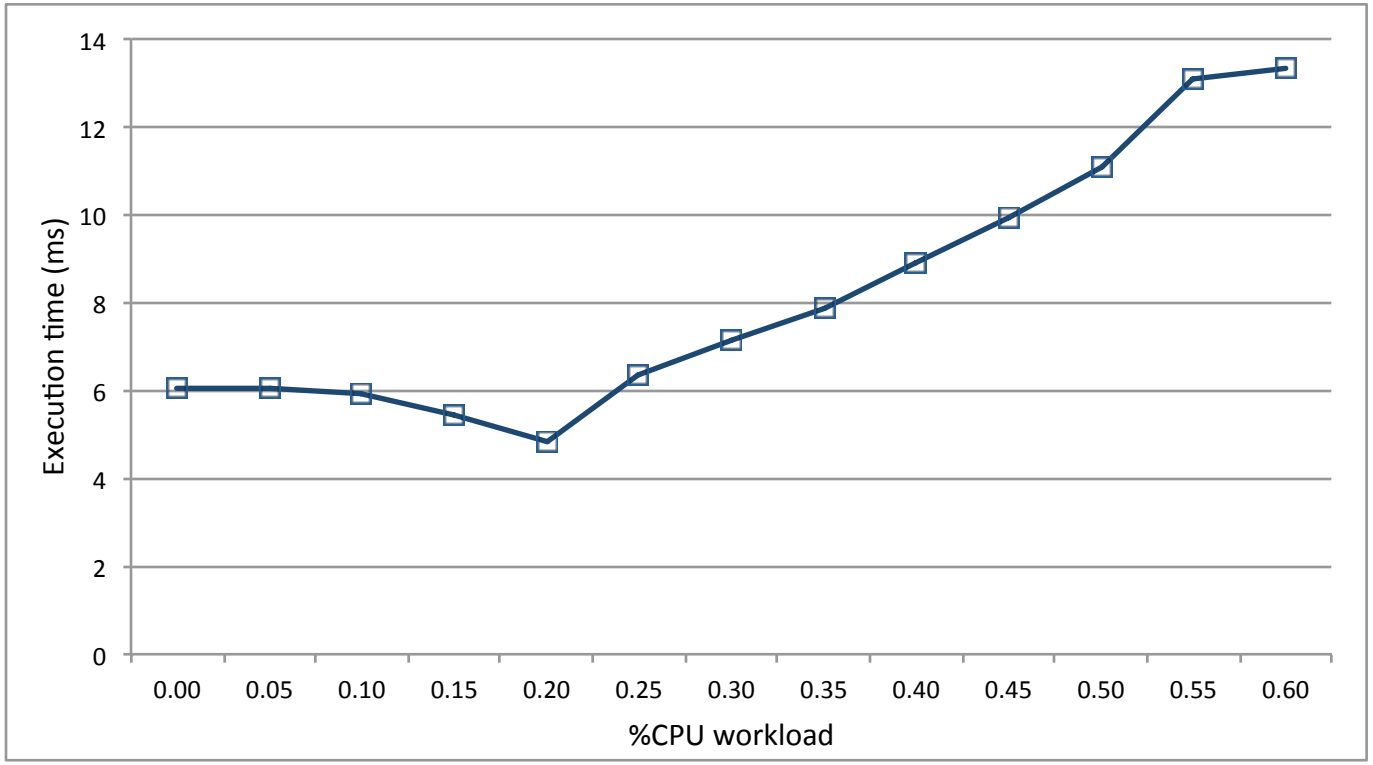
Stream Compaction (I)

- Stream compaction or filtering
  - Saving memory storage in sparse data
  - Similar to padding, but local reduction result (non-zero element count) is propagated



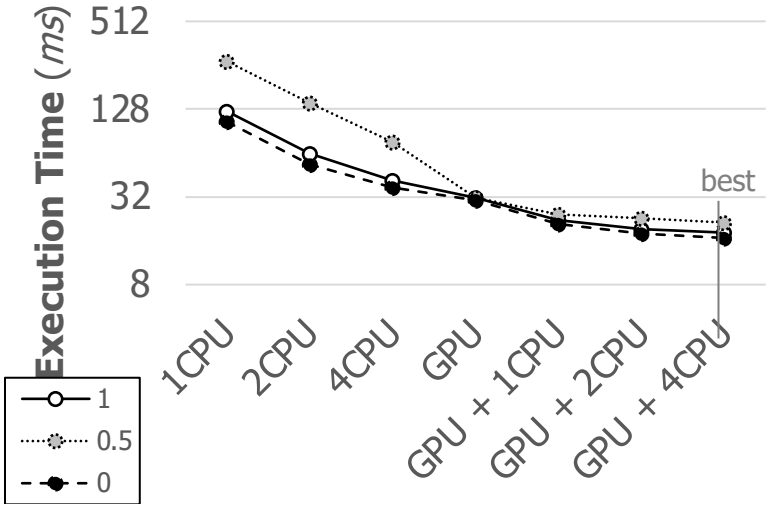
Stream Compaction (II)

- Performance results on NVIDIA Jetson TX1 (4 ARMv8 CPU cores + 2 GPU cores)
  - Array size: 2 MB, filtered items = 50%
  - 25% speedup over GPU only



Benefits of Collaboration: Stream Comp.

- AMD Kaveri (4 CPU cores + 8 GPU cores)
  - Data partitioning improves performance



Stream Compaction

(up to 82% improvement over GPU only)

Coarse-Grained Task Partitioning

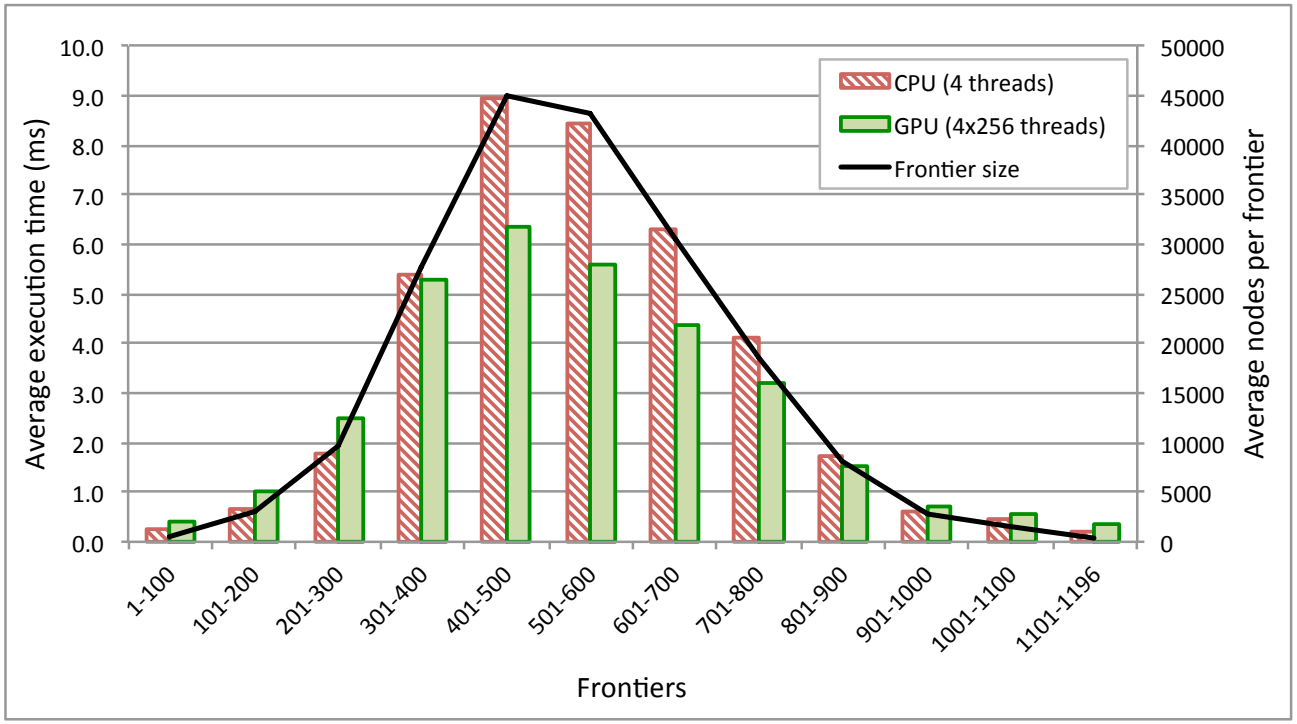
Breadth-First Search

- Small-sized and big-sized frontiers
  - Top-down approach
  - Kernel 1 and Kernel 2
- Atomic-based inter-block synchronization
  - Avoids kernel re-launch

- Very small frontiers
  - Underutilize GPU resources
- Collaborative implementation

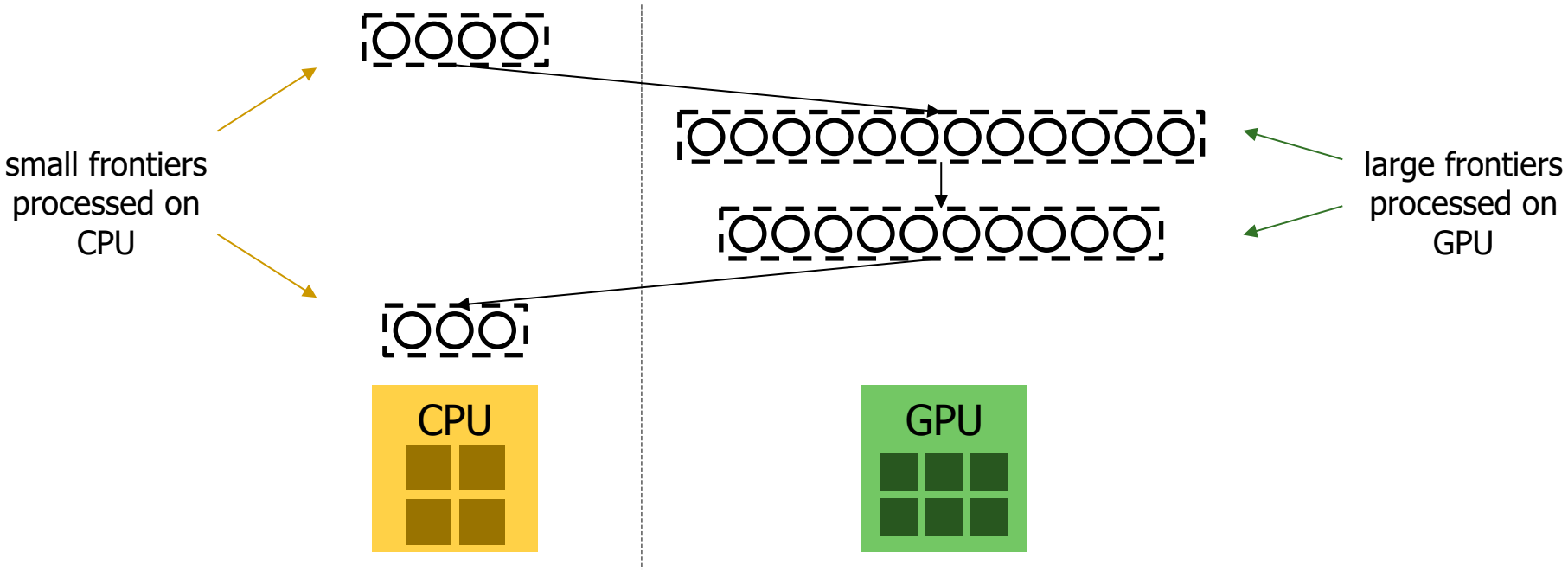
Recall: BFS on CPU or GPU?

- Motivation
  - Small-sized frontiers underutilize GPU resources
    - NVIDIA Jetson TX1 (4 ARMv8 CPUs + 2 SMXs)
    - New York City roads



BFS: Collaborative Implementation

- Choose the most appropriate device



Collaborative Implementation without UM

- Without Unified Memory (UM)
  - Explicit memory copies

```
// Host code
while(frontier_size != 0){

    if(frontier_size < LIMIT){

        // Launch CPU threads

    }
    else{

        // Copy from host to device (queues and synchronization variables)

        // Launch GPU kernel

        // Copy from device to host (queues and synchronization variables)

    }

}
```

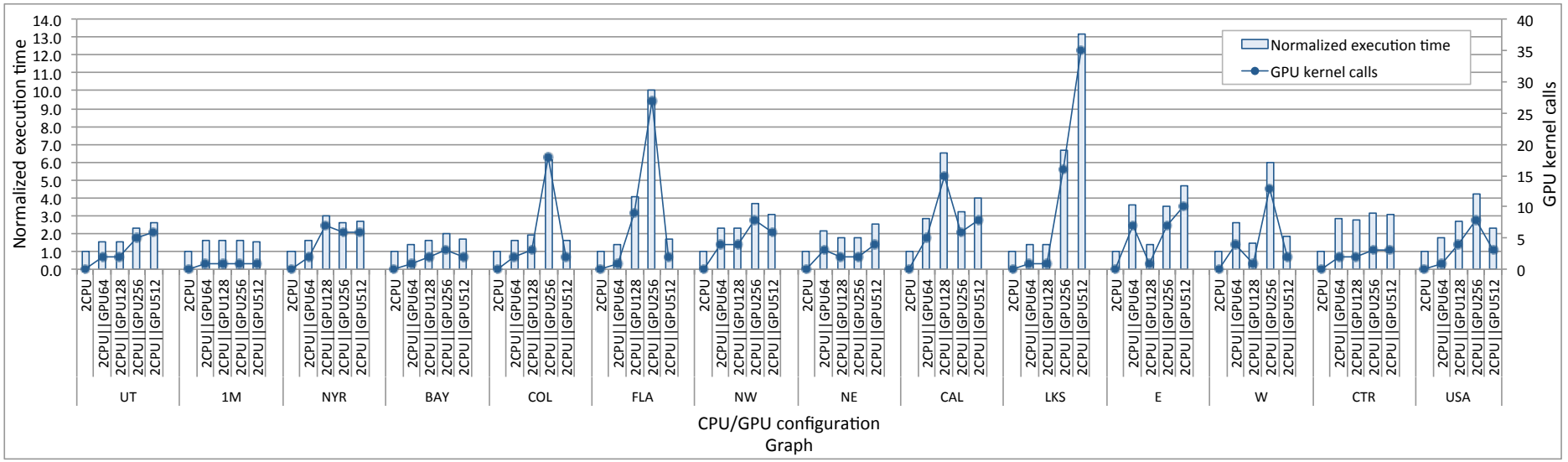
Collaborative Implementation with UM (I)

- Unified Memory
  - `cudaMallocManaged();`
  - Easier programming
  - No explicit memory copies

```
// Host code
while(frontier_size != 0){

    if(frontier_size < LIMIT){

        // Launch CPU threads

    }
    else{

        // Launch GPU kernel for every frontier (kernel termination and relaunch)

        cudaDeviceSynchronize();

    }

}
```

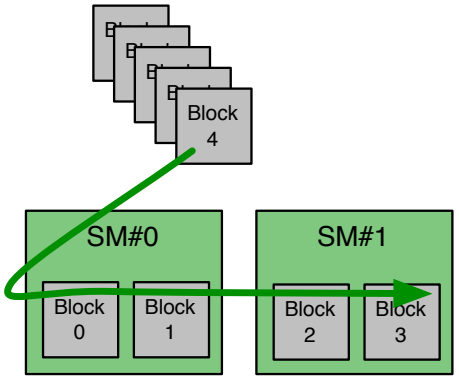
BFS: Kernel Termination and Relaunch

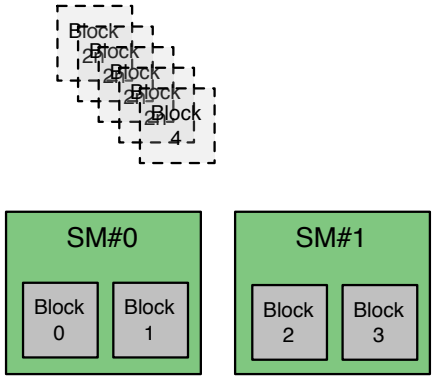
- AMD Kaveri (4 CPU cores + 8 GPU cores)
  - High overhead of kernel relaunch makes CPU+GPU collaboration impractical

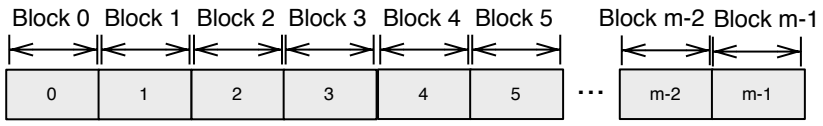


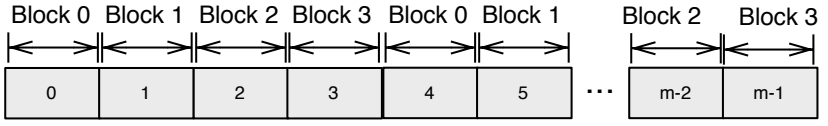
Recall: Persistent Thread Blocks

- Combine Kernel 1 and Kernel 2
- We can avoid kernel re-launch
- We need to use persistent thread blocks
  - Kernel 2 launches (frontier_size / block_size) blocks
  - Persistent blocks: up to (number_SMs x max_blocks_SM)









Atomic-based Block Synchronization (I)

- Code (simplified)

```
// GPU kernel
const int gtid = blockIdx.x * blockDim.x + threadIdx.x;

while(frontier_size != 0){

    for(node = gtid; node < frontier_size; node += blockDim.x * gridDim.x){

      // Visit neighbors
      // Enqueue in output queue if needed (global or local queue)

    }

    // Update frontier_size

    // Global synchronization
}
```

Atomic-based Block Synchronization (II)

- Global synchronization (simplified)
  - At the end of each iteration

```
const int tid = threadIdx.x;
const int gtid = blockIdx.x * blockDim.x + threadIdx.x;
atomicExch(ptr_threads_run, 0);
atomicExch(ptr_threads_end, 0);
int frontier = 0;
 ...
frontier++;

if(tid == 0){
    atomicAdd(ptr_threads_end, 1);  // Thread block finishes iteration
}

if(gtid == 0){
    while(atomicAdd(ptr_threads_end, 0) != gridDim.x){;}  // Wait until all blocks finish

    atomicExch(ptr_threads_end, 0);  // Reset
    atomicAdd(ptr_threads_run, 1);  // Count iteration
}

if(tid == 0 && gtid != 0){
    while(atomicAdd(ptr_threads_run, 0) < frontier){;}  // Wait until ptr_threads_run is updated
}

__syncthreads();  // Rest of threads wait here

...
```

BFS: Collaborative Implementation (II)

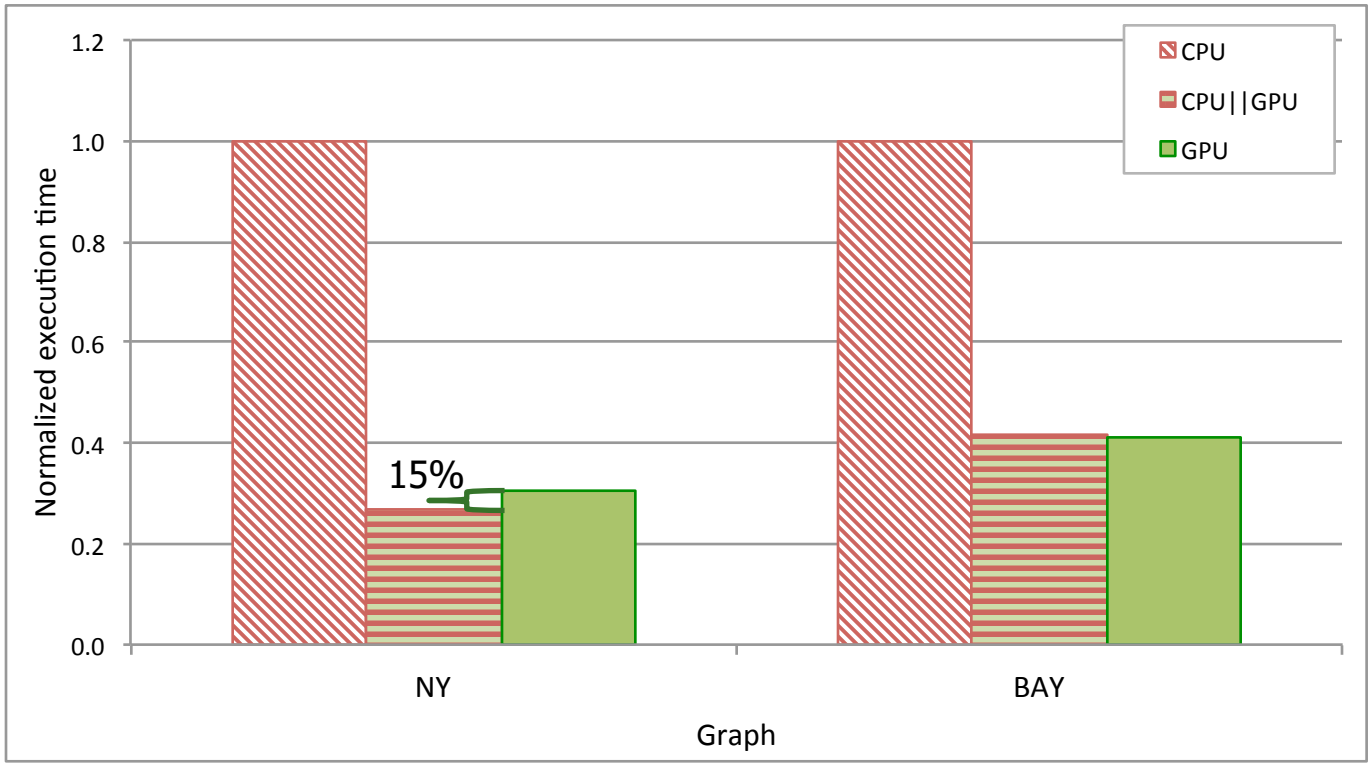
- Choose CPU or GPU depending on frontier

```
// Host code
while(frontier_size != 0){

    if(frontier_size < LIMIT){

        // Launch CPU threads
    }
    else{

        // Launch GPU kernel (keep running while frontier_size >= LIMIT)

        cudaDeviceSynchronize();
    }
}
```

- CPU threads or GPU kernel keep running while the condition is satisfied

BFS: Collaborative Implementation (III)

- Experimental results
  - NVIDIA Jetson TX1 (4 ARMv8 CPU cores + 2 GPU cores)



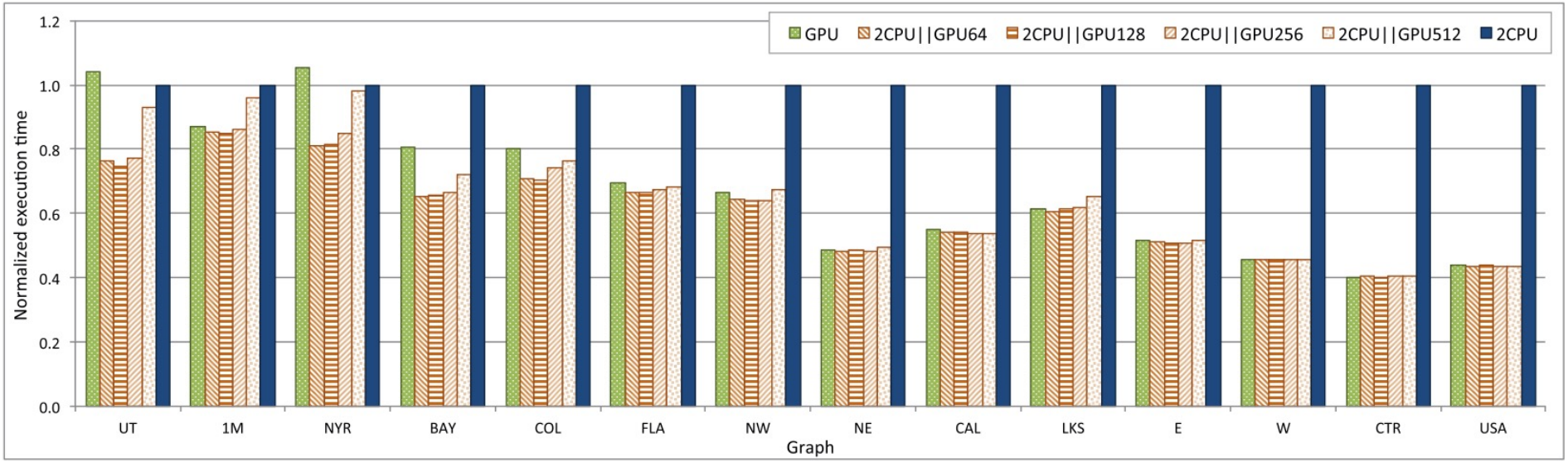
Collaborative Implementation with UM (II)

- Pascal/Volta/Turing/Ampere Unified Memory & HSA
  - CPU/GPU coherence
  - System-wide atomic operations
  - No need to re-launch kernel or CPU threads
  - Possibility of CPU and GPU working on the same frontier

```
// Host code
while(frontier_size != 0){

    if(frontier_size < LIMIT){

        // Launch CPU threads (compute when frontier_size < LIMIT)
    }
    else{

        // Launch GPU kernel (compute when frontier_size >= LIMIT)
    }
}
cudaDeviceSynchronize();
```

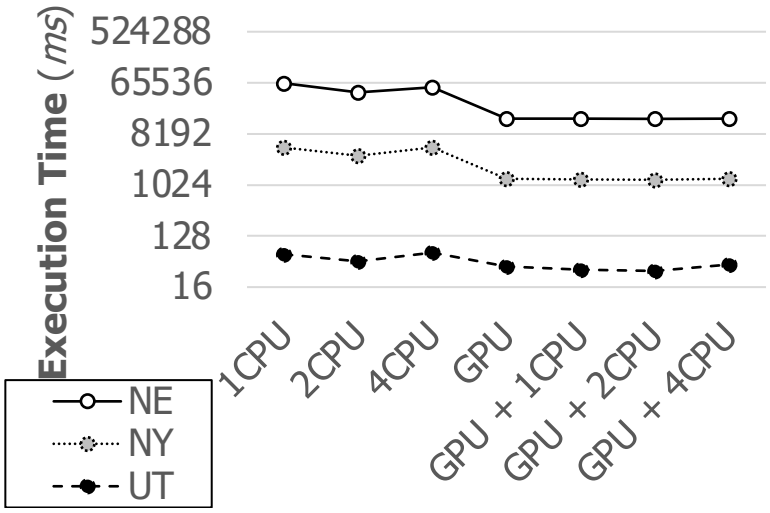
Benefits of Collaboration: BFS

- AMD Kaveri (4 CPU cores + 8 GPU cores)
  - The collaborative implementation (with system-wide atomics) is up to 39% faster than the GPU only version



Benefits of Collaboration: SSSP

- AMD Kaveri (4 CPU cores + 8 GPU cores)
  - SSSP performs more computation than BFS



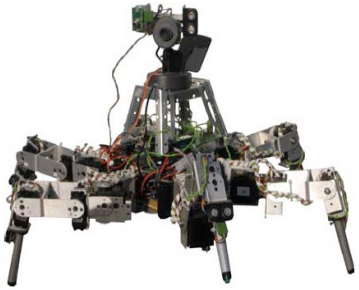
Single Source Shortest Path

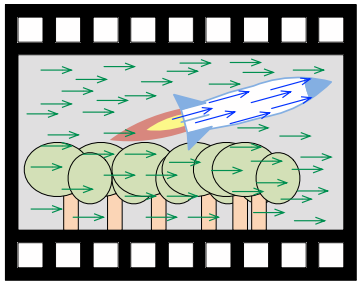
(up to 22% improvement over GPU only)

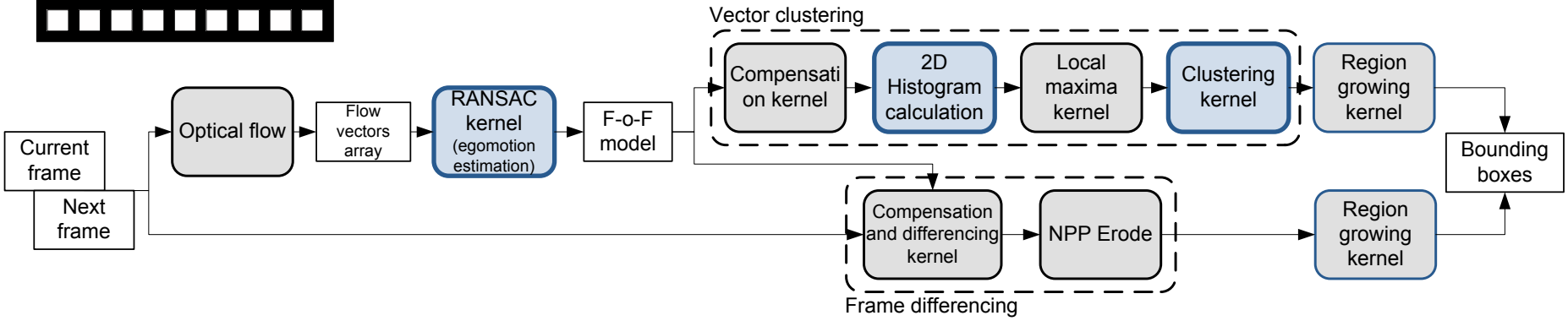
Fine-Grained Task Partitioning

Egomotion Compensation and Moving Objects Detection (I)

- Hexapod robot OSCAR
  - Rescue scenarios
  - Strong egomotion on uneven terrains
- Algorithm
  - Random Sample Consensus (RANSAC): F-o-F model

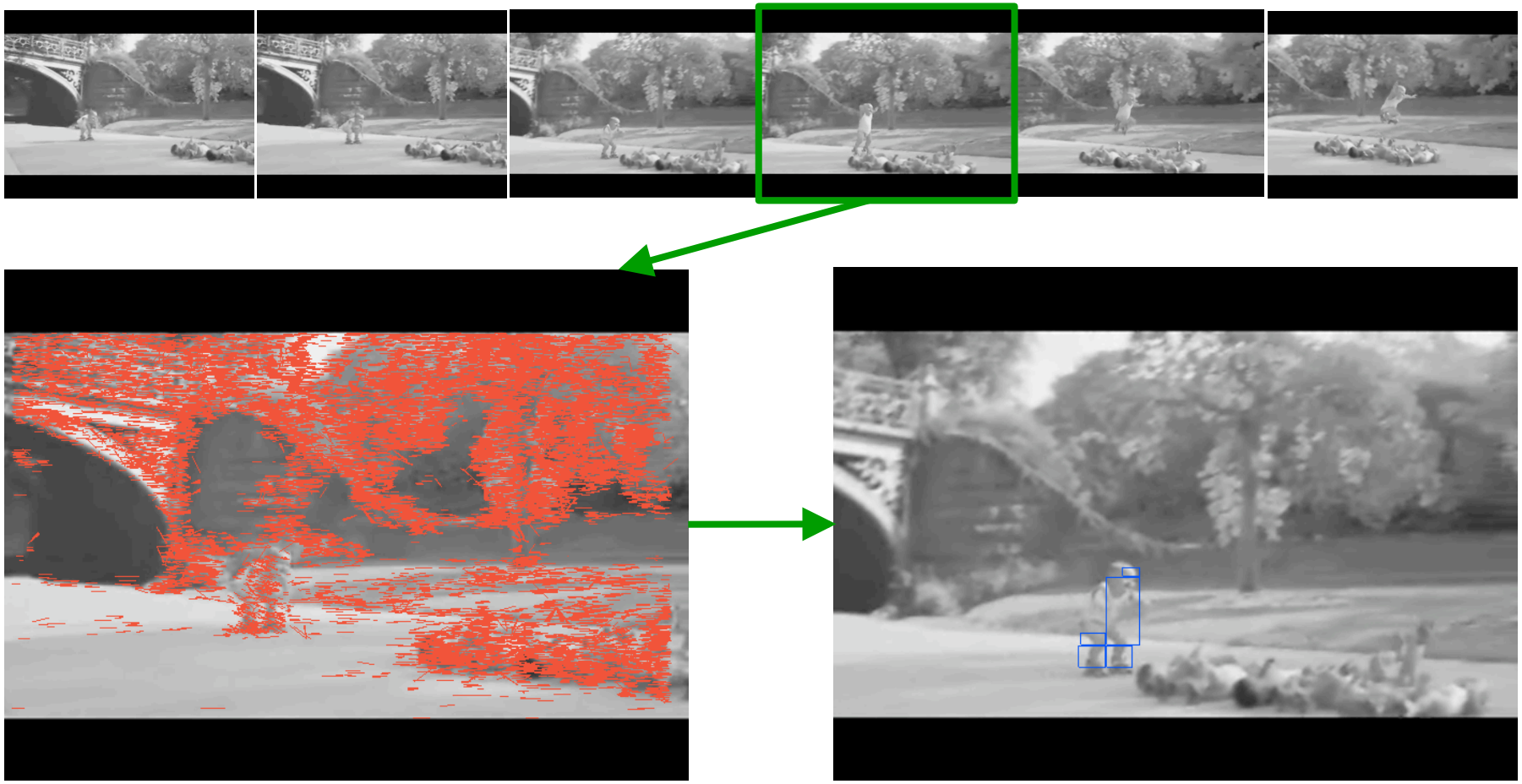






Egomotion Compensation and Moving Objects Detection (II)

Fast moving object in strong egomotion scenario detected by vector clustering

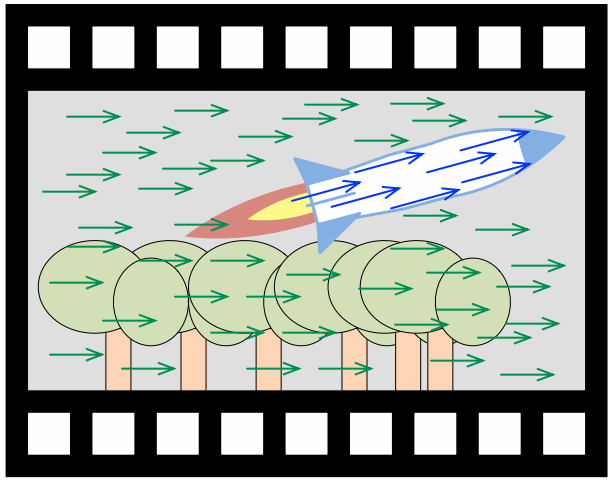


RANSAC: SISD and SIMD Phases

- RANSAC (Fischler+, 1981)

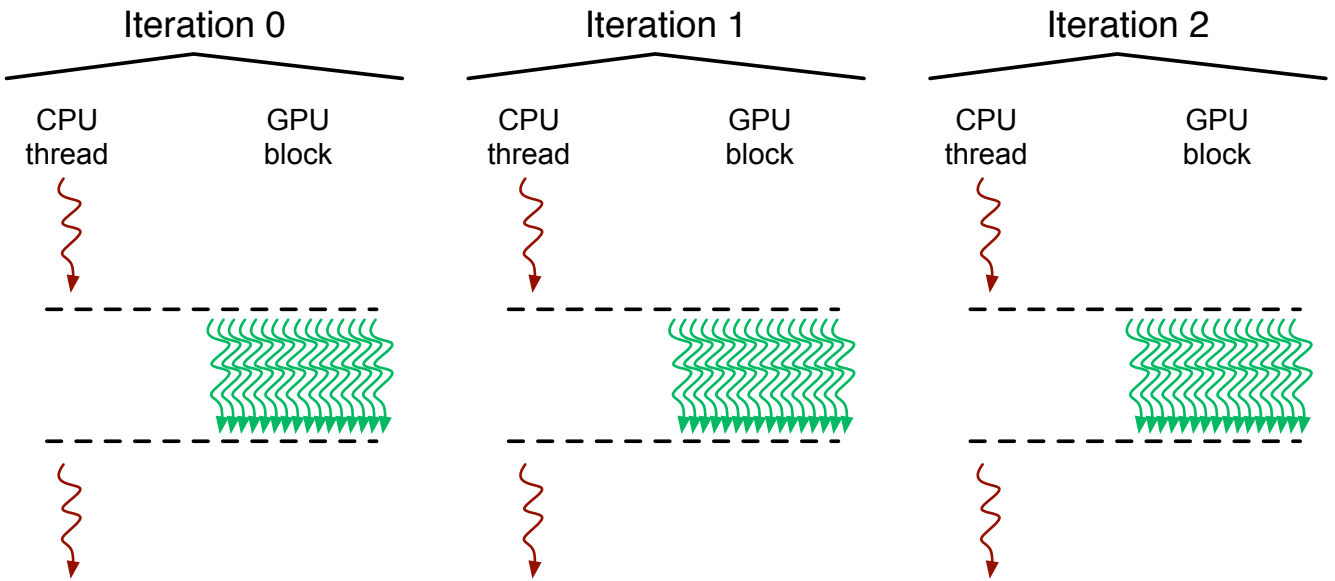
```
while (iteration < MAX_ITER){
     Fitting stage (Compute F-o-F model)          // SISD phase
     Evaluation stage (Count outliers)            // SIMD phase
     Comparison to best model                     // SISD phase
     Check if best model is good enough and iteration >= MIN_ITER     // SISD phase
}
```

  - Fitting stage picks two flow vectors randomly
  - Evaluation generates motion vectors from F-o-F model, and compares them to real flow vectors

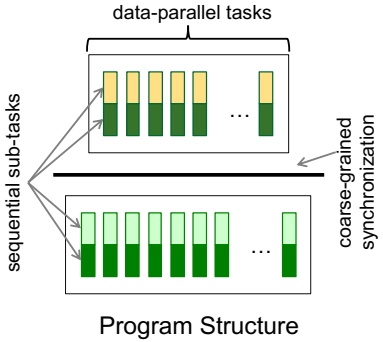


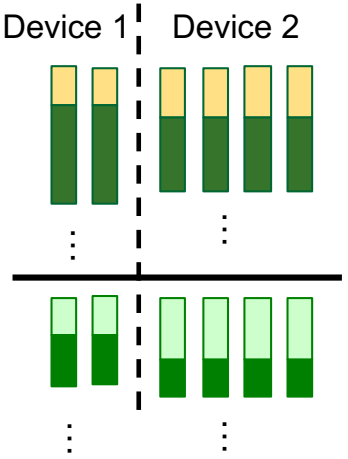
Collaborative Implementation

- Randomly picked vectors: Iterations are independent
  - We assign one iteration to one CPU thread and one GPU block

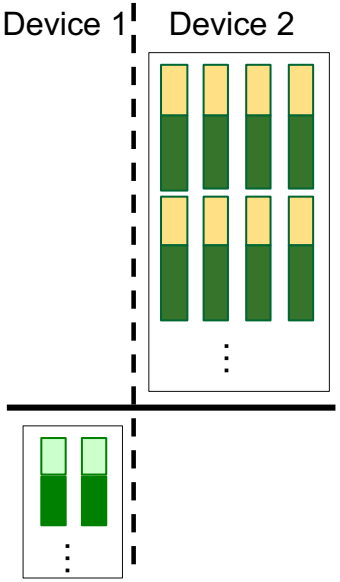


Collaborative Patterns

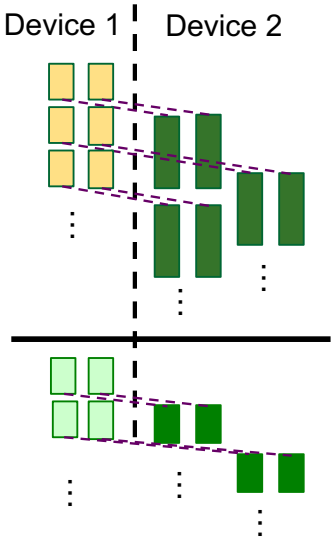




Data Partitioning



Coarse-grained Task Partitioning



Fine-grained Task Partitioning

Chai Benchmark Suite

- Collaborative Heterogeneous Applications for Integrated architectures
- Heterogeneous execution on CPU, GPU, FPGA
- Collaboration patterns
  - 8 data partitioning benchmarks
  - 3 coarse-grain task partitioning benchmarks
  - 3 fine-grain task partitioning benchmarks
- Discrete (D) and Unified (U) versions
  - CUDA, OpenCL, and C++AMP for CPU+GPU
  - OpenCL for CPU+FPGA
  - CUDA-Sim for Gem5-GPU

https://chai-benchmarks.github.io

Chai Benchmarks

Collaboration Pattern		Short Name	Benchmark
Data Partitioning		BS	Bézier Surface
		CEDD	Canny Edge Detection
		HSTI	Image Histogram (Input Partitioning)
		HSTO	Image Histogram (Output Partitioning)
		PAD	Padding
		RSCD	Random Sample Consensus
		SC	Stream Compaction
		TRNS	In-place Transposition
Task Partitioning	Fine-grain	RSCT	Random Sample Consensus
		TQ	Task Queue System (Synthetic)
		TQH	Task Queue System (Histogram)
	Coarse-grain	BFS	Breadth-First Search
		CEDT	Canny Edge Detection
		SSSP	Single-Source Shortest Path

Versions:

- OpenCL-**U**
- OpenCL-**D**
- CUDA-**U**
- CUDA-**D**
- CUDA-**U**-Sim
- CUDA-**D**-Sim
- C++AMP

Chai: Diversity of Benchmarks (I)

- Diversity of partitioning, usage of system-wide atomics, load balancing, and concurrency

DATA PARTITIONING

Benchmark	Partitioning Granularity	Partitioned Data	System-wide Atomics	Load Balance
BS	Fine	Output	None	Yes
CEDD	Coarse	Input, Output	None	Yes
HSTI	Fine	Input	Compute	No
HSTO	Fine	Output	None	No
PAD	Fine	Input, Output	Sync	Yes
RSCD	Medium	Output	Compute	Yes
SC	Fine	Input, Output	Sync	No
TRNS	Medium	Input, Output	Sync	No

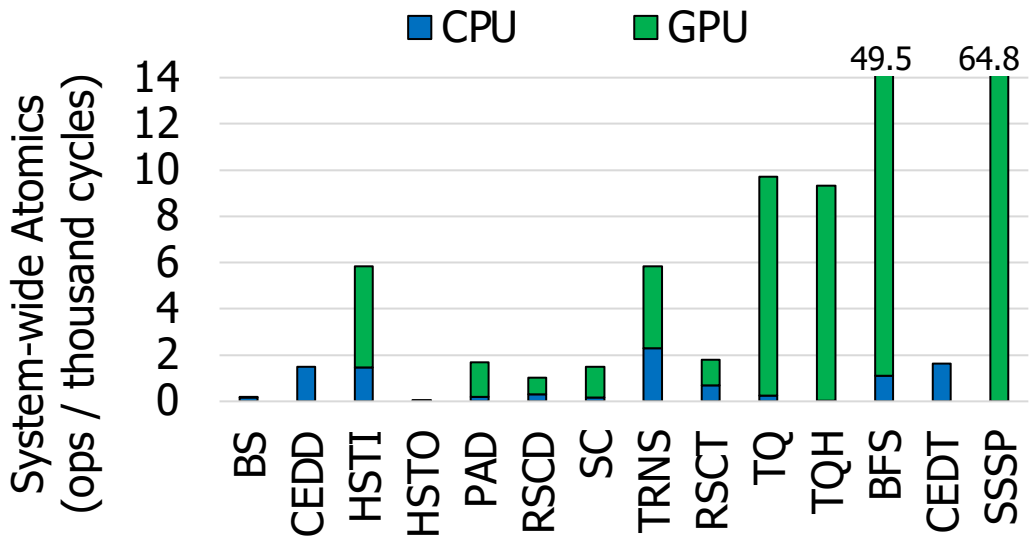
FINE-GRAIN TASK PARTITIONING

Benchmark	System-wide Atomics	Load Balance
RSCT	Sync, Compute	Yes
TQ	Sync	No
TQH	Sync	No

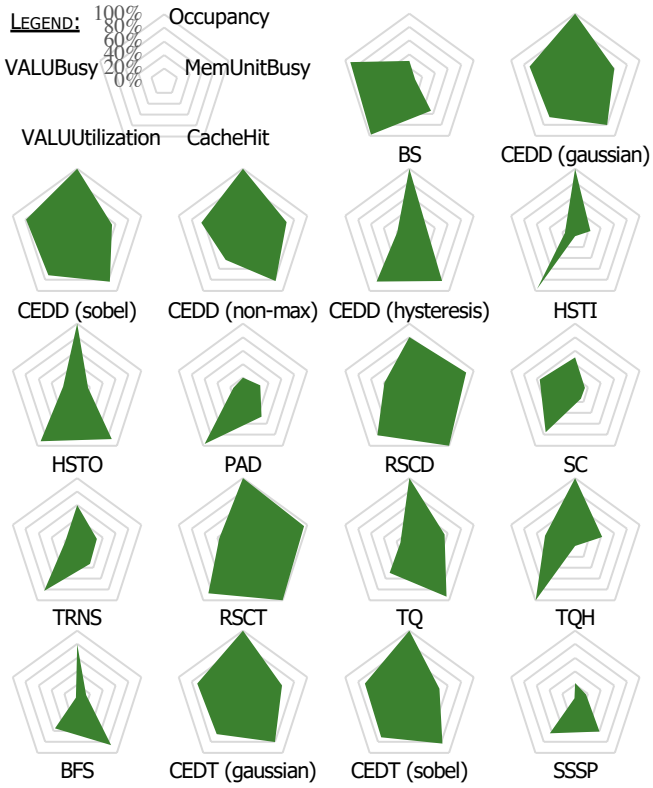
COARSE-GRAIN TASK PARTITIONING

Benchmark	System-wide Atomics	Partitioning	Concurrency
BFS	Sync, Compute	Iterative	No
CEDT	Sync	Non-iterative	Yes
SSSP	Sync, Compute	Iterative	No

Chai: Diversity of Benchmarks (II)

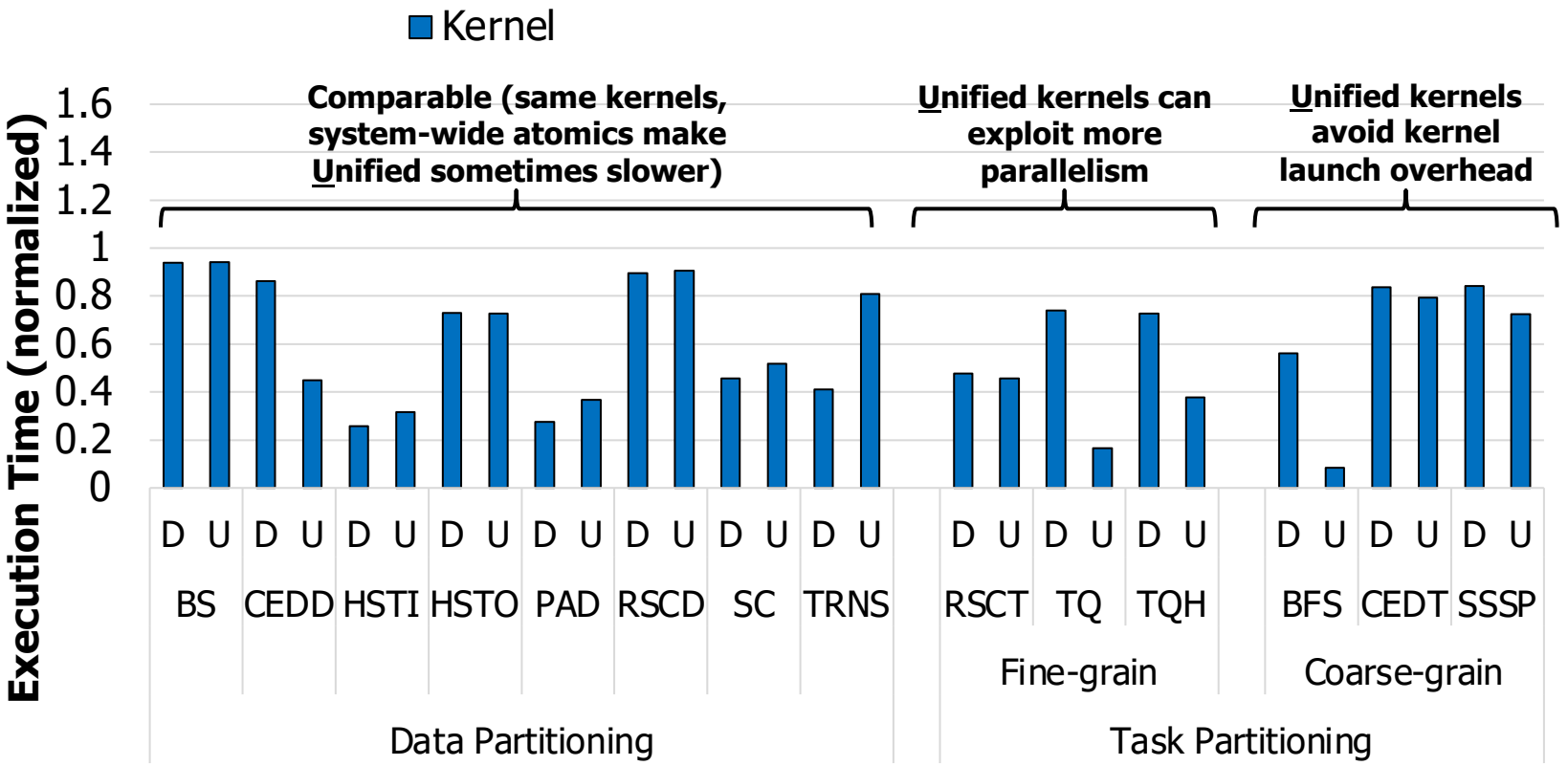


Varying intensity in use of system-wide atomics



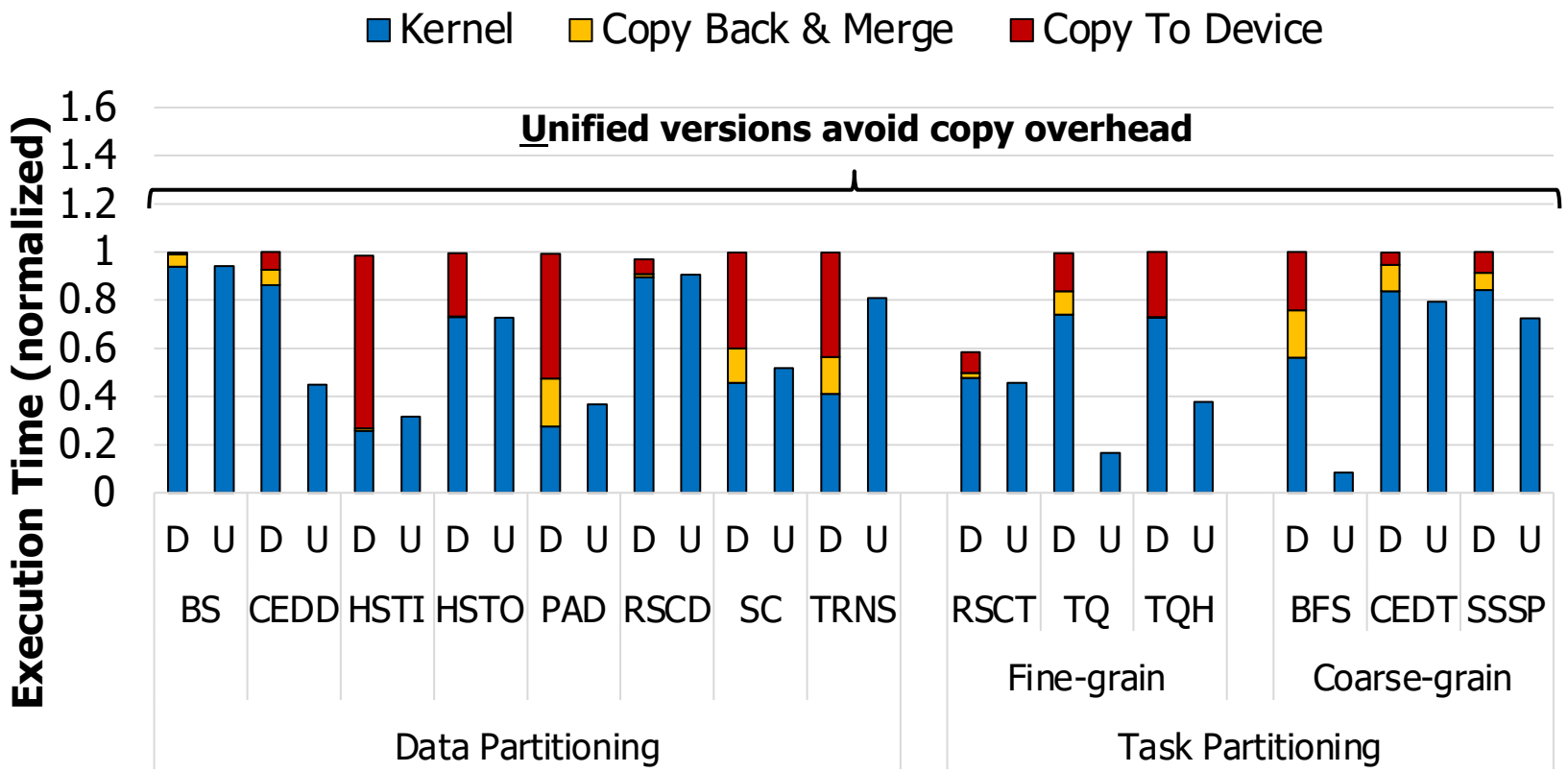
Diverse execution profiles

Benefits of Unified Memory: Kernel Time



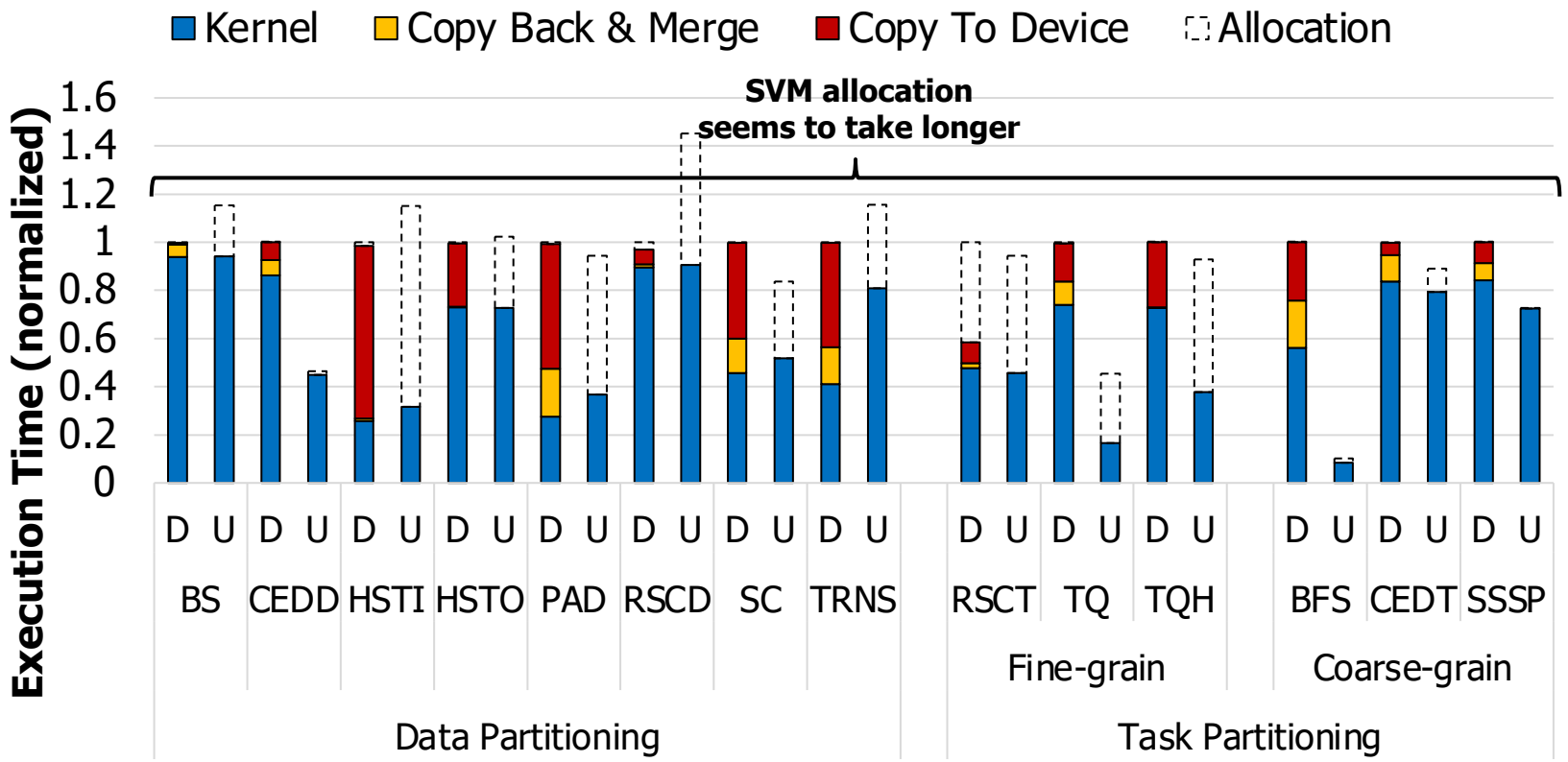
AMD Kaveri (4 CPU cores + 8 GPU cores), OpenCL

Benefits of Unified Memory: Data Transfers



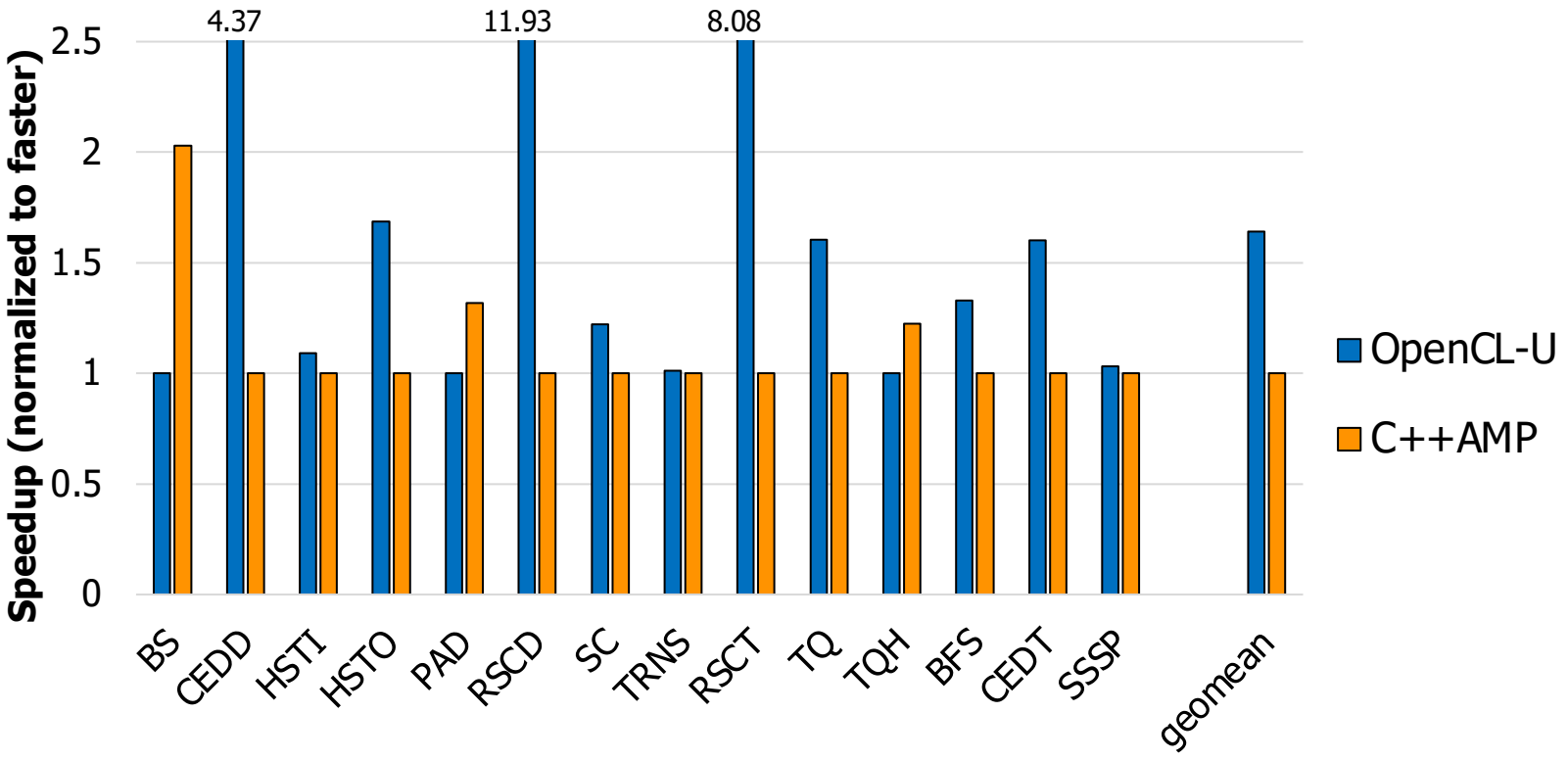
AMD Kaveri (4 CPU cores + 8 GPU cores), OpenCL

Benefits of Unified Memory: Allocation



AMD Kaveri (4 CPU cores + 8 GPU cores), OpenCL

Comparison C++AMP vs. OpenCL-U

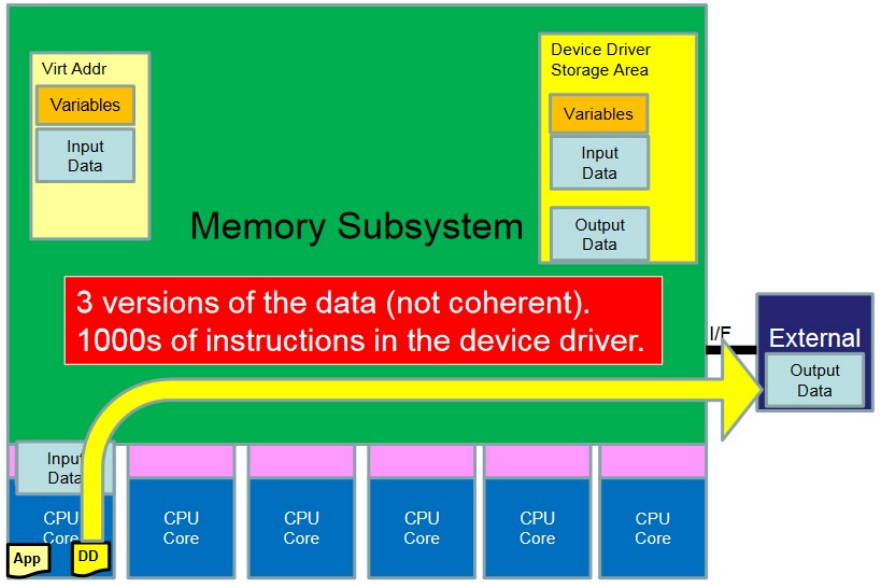


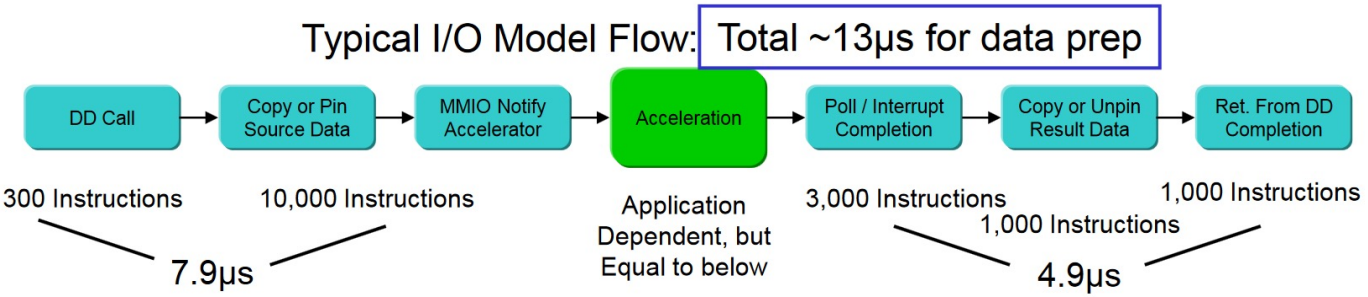
Heterogeneous System Architecture

- Wen-mei W. Hwu (editor), "Heterogeneous System Architecture: A New Compute Platform Infrastructure," 2016
  - Chapter 8 – Application

use cases: Platform atomics

Background: Traditional I/O Technology

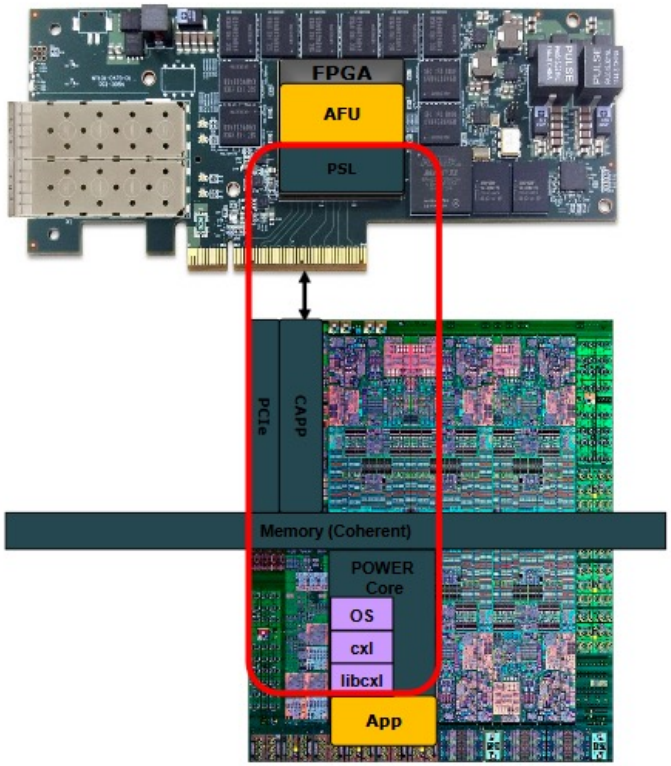




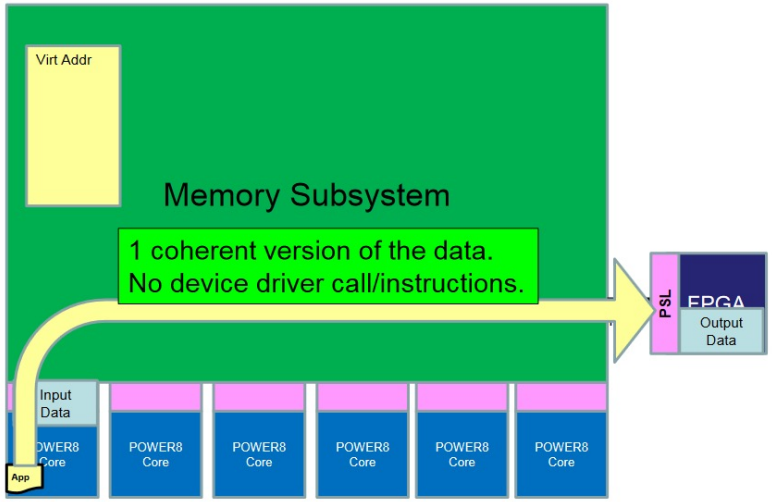
Dionysios Diamantopoulos, IBM Research – Zurich, COOL Chips 2018

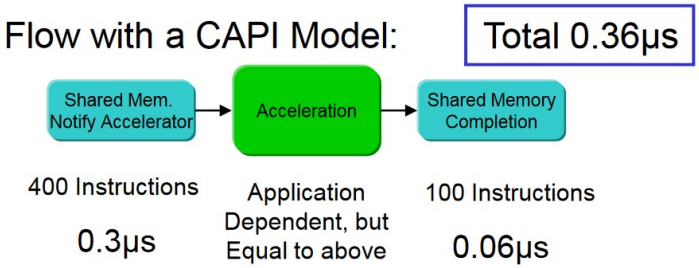
CAPI/OpenCAPI Overview

- CAPI/CAPI2 (Coherent Accelerator Processor Interface)
- OpenCAPI



POWER8 – POWER9 Processor



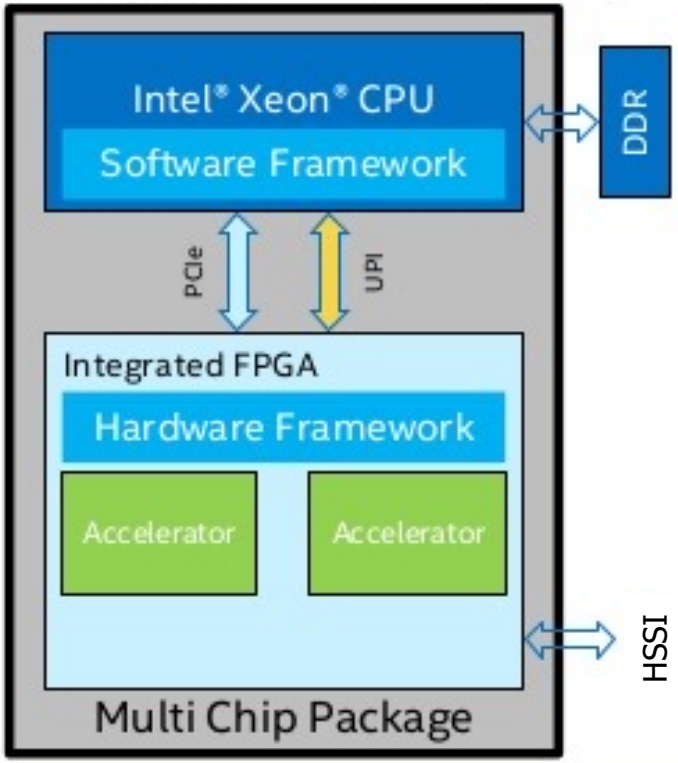


Dionysios Diamantopoulos, IBM Research – Zurich, COOL Chips 2018

Collaborative Computing on CPU+FPGA

- Traditionally, accelerators (GPUs, FPGAs, etc.) have been used as *offload* engines
- Heterogeneous architectures moving towards tighter integration
  - Unified memory
  - System-wide atomics
- Tighter integration allows fine-grained collaboration

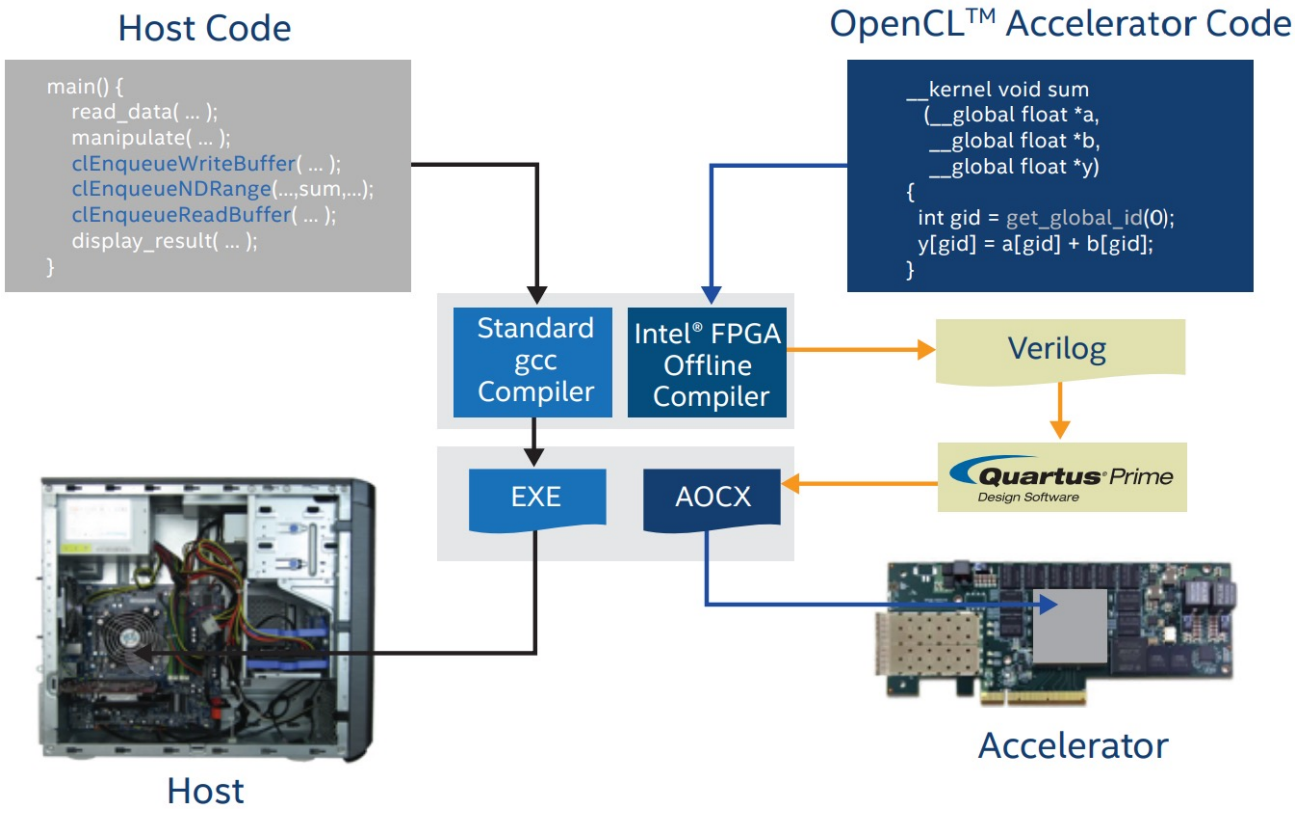
Key challenge: identify the best CPU-FPGA collaboration strategy



Intel Xeon + FPGA Integrated Platform (MCP)

Intel OpenCL SDK for FPGA

- Intel OpenCL SDK for FPGA is used to compile and synthesize host executable and FPGA design



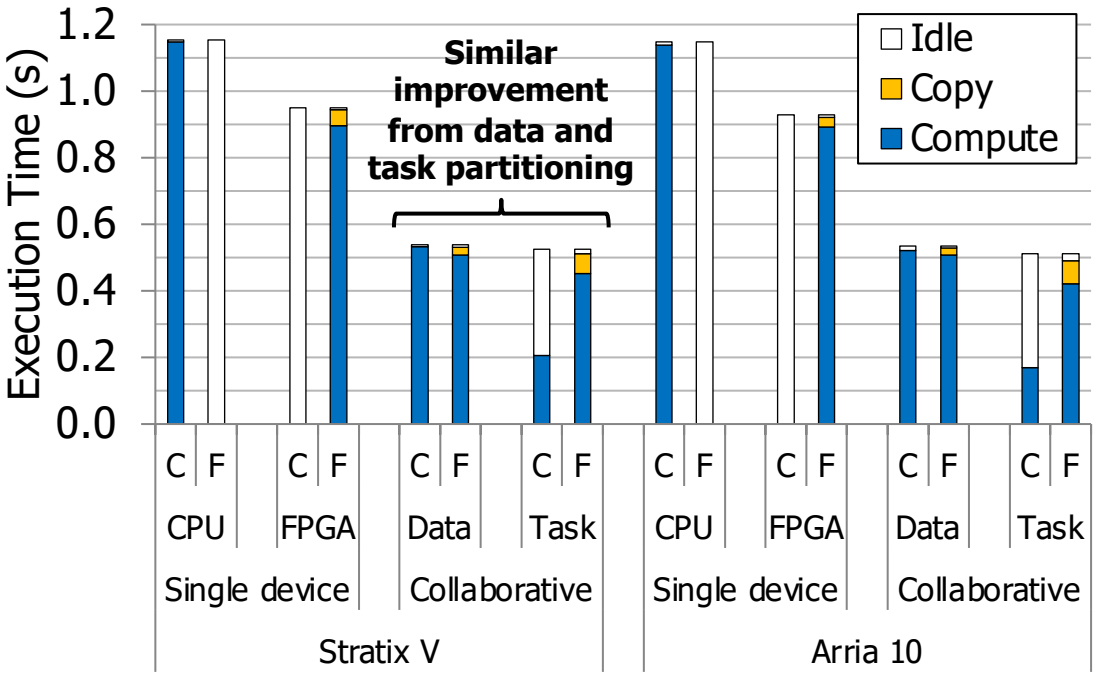
CPU+FPGA Evaluation Platforms

	Platform A	Platform B
FPGA Board	Terasic DE5-Net	Nallatech 510T
FPGA Chip	**Intel Stratix V GX**	**Intel Arria 10 GX**
On-Board Memory	4 GB (DDR3)	8 GB (DDR4)
Host CPU	Intel Xeon E3-1240 v3	Intel Xeon E5-2650 v3
Host Memory	8 GB (DDR3)	96 GB (DDR4)
Interface	PCIe gen3.0 x8	PCIe gen3.0 x8

Huang+, "Analysis and Modeling of Collaborative Execution Strategies for Heterogeneous CPU-FPGA Architectures," ICPE 2019

Benefits of Collaboration on FPGA (I)

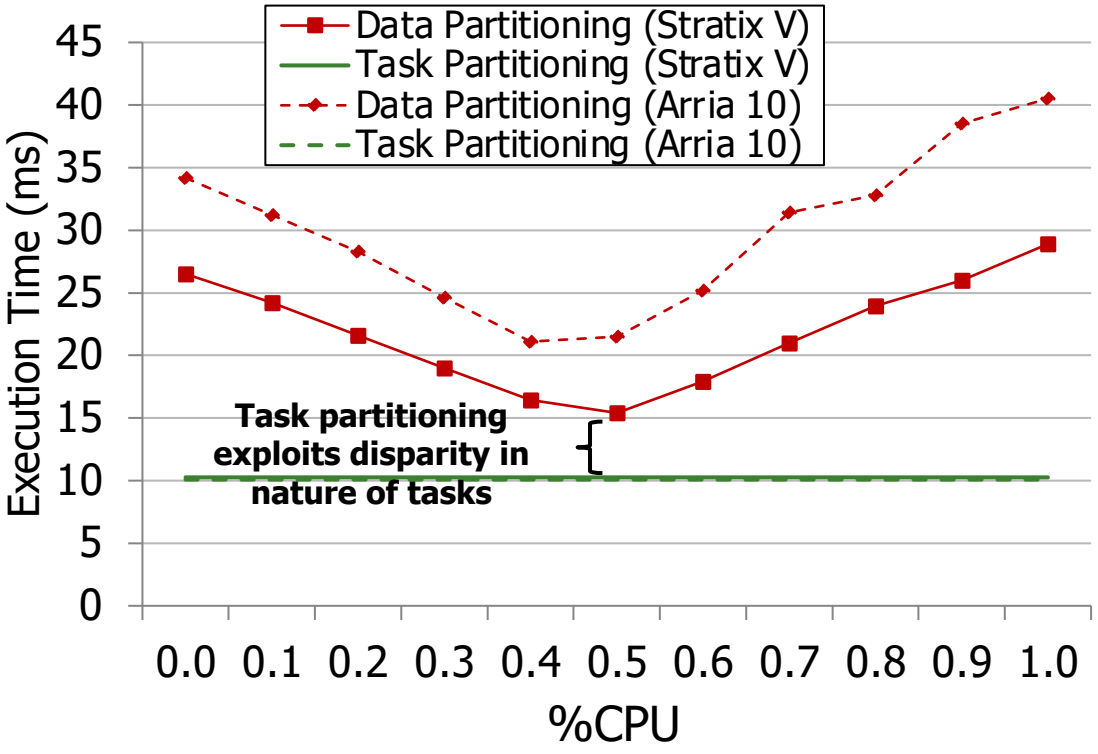
Case Study:
Canny Edge Detection



Chang+, "Collaborative Computing for Heterogeneous Integrated Systems," ICPE 2017
Huang+, "Analysis and Modeling of Collaborative Execution Strategies for Heterogeneous CPU-FPGA Architectures," ICPE 2019

Benefits of Collaboration on FPGA (II)

Case Study:
Random
Sample
Consensus



Chang+, "Collaborative Computing for Heterogeneous Integrated Systems," ICPE 2017
Huang+, "Analysis and Modeling of Collaborative Execution Strategies for Heterogeneous CPU-FPGA Architectures," ICPE 2019

Chai on CPU-FPGA Systems (I)

- Sitao Huang, Li-Wen Chang, Izzat El Hajj, Simon Garcia De Gonzalo, Juan Gomez-Luna, Sai Rahul Chalamalasetti, Mohamed El-Hadedy, Dejan Milojicic, Onur Mutlu, Deming Chen, and Wen-mei Hwu,
  "Analysis and Modeling of Collaborative Execution Strategies for Heterogeneous CPU-FPGA Architectures"
  *Proceedings of the 10th ACM/SPEC International Conference on Performance Engineering (**ICPE**)*, Mumbai, India, April 2019.
  [Slides (pptx) (pdf)]
  [Chai CPU-FPGA Benchmark Suite]

Analysis and Modeling of Collaborative Execution Strategies for Heterogeneous CPU-FPGA Architectures

Sitao Huang
ECE, UIUC
shuang91@illinois.edu

Li-Wen Chang*
Microsoft
liwen.chang@microsoft.com

Izzat El Hajj
ECE, UIUC
elhajj2@illinois.edu

Simon Garcia De Gonzalo
CS, UIUC
grcdgnz2@illinois.edu

Juan Gómez-Luna
CS, ETH Zurich
juang@ethz.ch

Sai Rahul Chalamalasetti
Hewlett Packard Labs
sairahul.chalamalasetti@hpe.com

Mohamed El-Hadedy
ECE, Cal Poly Pomona
mealy@cpp.edu

Dejan Milojicic
Hewlett Packard Labs
dejan.milojicic@hpe.com

Onur Mutlu
CS, ETH Zurich
omutlu@ethz.ch

Deming Chen
ECE, UIUC
dchen@illinois.edu

Wen-mei Hwu
ECE, UIUC
w-hwu@illinois.edu

Chai on CPU-FPGA Systems (II)

- Jiantong Jiang, Zeke Wang, Xue Liu, Juan Gómez-Luna, Nan Guan, Qingxu Deng, Wei Zhang, and Onur Mutlu,
  "Boyi: A Systematic Framework for Automatically Deciding the Right Execution Model of OpenCL Applications on FPGAs"
  *Proceedings of the 28th International Symposium on Field-Programmable Gate Arrays (**FPGA**)*, Seaside, CA, USA, February 2020.
  [Slides (pptx) (pdf)]

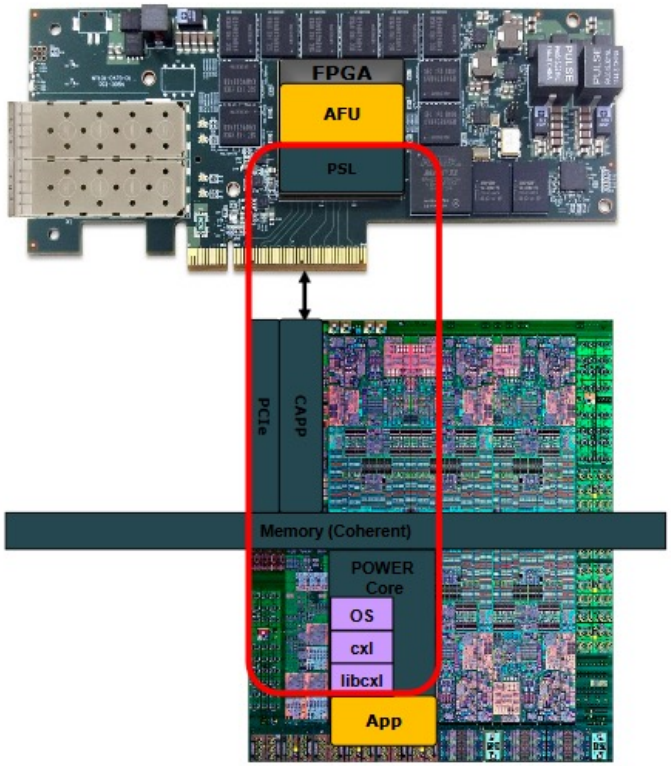
Boyi: A Systematic Framework for Automatically Deciding the Right Execution Model of OpenCL Applications on FPGAs

Jiantong Jiang[1⋆] Zeke Wang[2⋆] Xue Liu[1*] Juan Gómez-Luna[2]
Nan Guan[3] Qingxu Deng[1] Wei Zhang[4] Onur Mutlu[2]

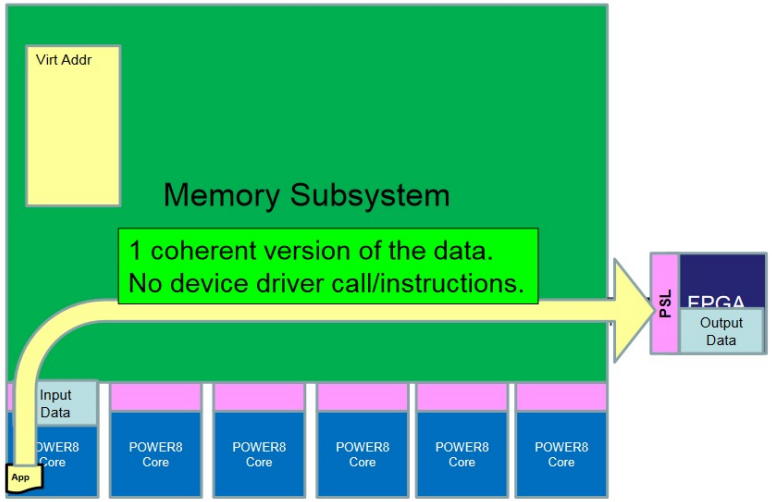
[1] Department of Computer Science and Engineering, Northeastern University, China
[2] ETH Zürich, Switzerland
[3] Department of Computing, Hong Kong Polytechnic University, Hong Kong
[4] Department of Electronic and Computer Engineering, Hong Kong University of Science and Technology, Hong Kong

CAPI/OpenCAPI Overview

- CAPI/CAPI2 (Coherent Accelerator Processor Interface)
- OpenCAPI



POWER8 – POWER9 Processor



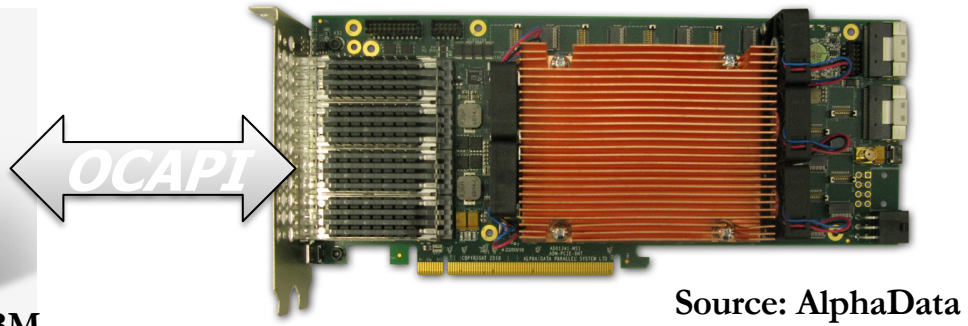
Flow with a CAPI Model: Total 0.36µs

400 Instructions	Application Dependent, but Equal to above	100 Instructions
0.3µs		0.06µs

Dionysios Diamantopoulos, IBM Research – Zurich, COOL Chips 2018

Evaluation Setup for Weather Acceleration

POWER9 AC922

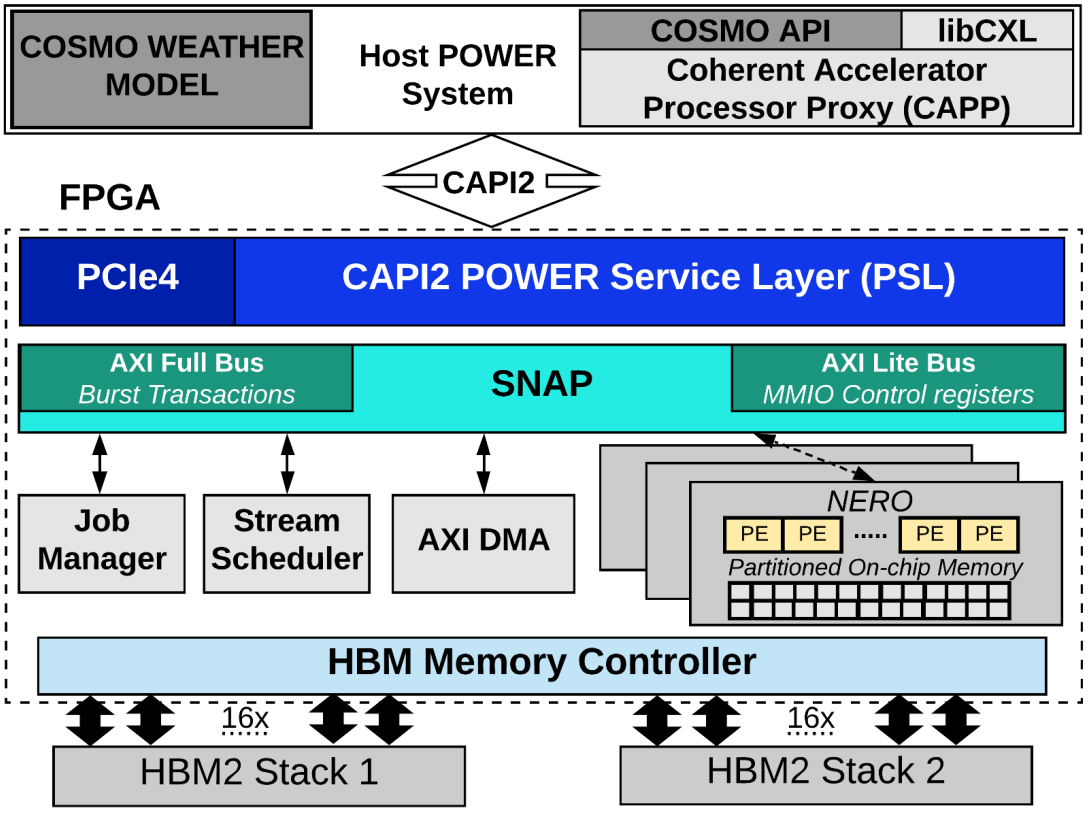


HBM-based AD9H7 board

- **Host System**
  IBM POWER9-16 core (64-threads)
- **FPGA board**
  Xilinx Virtex® Ultrascale+™ XCVU37P-2

NERO Application Framework

- NERO communicates to Host over CAPI2 (Coherent Accelerator Processor Interface)
- COSMO API handles offloading jobs to NERO
- SNAP (Storage, Network, and Analytics Programming) allows for seamless integration of the COSMO API



https://github.com/open-power/snap

Accelerating Climate Modeling (I)

- Gagandeep Singh, Dionysios Diamantopoulos, Christoph Hagleitner, Juan Gómez-Luna, Sander Stuijk, Onur Mutlu, and Henk Corporaal,
  "NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling"
  *Proceedings of the 30th International Conference on Field-Programmable Logic and Applications (**FPL**)*, Gothenburg, Sweden, September 2020.
  [Slides (pptx) (pdf)]
  [Lightning Talk Slides (pptx) (pdf)]
  [Talk Video (23 minutes)]
  Nominated for the Stamatis Vassiliadis Memorial Award.

NERO: A Near High-Bandwidth Memory Stencil Accelerator for Weather Prediction Modeling

Gagandeep Singh[a,b,c] Dionysios Diamantopoulos[c] Christoph Hagleitner[c] Juan Gómez-Luna[b]
Sander Stuijk[a] Onur Mutlu[b] Henk Corporaal[a]
[a]Eindhoven University of Technology [b]ETH Zürich [c]IBM Research Europe, Zurich

Accelerating Climate Modeling (II)

- Gagandeep Singh, Mohammed Alser, Damla Senol Cali, Dionysios Diamantopoulos, Juan Gómez-Luna, Henk Corporaal, and Onur Mutlu, **"FPGA-based Near-Memory Acceleration of Modern Data-Intensive Applications"**
  IEEE Micro (***IEEE MICRO***), 2021.

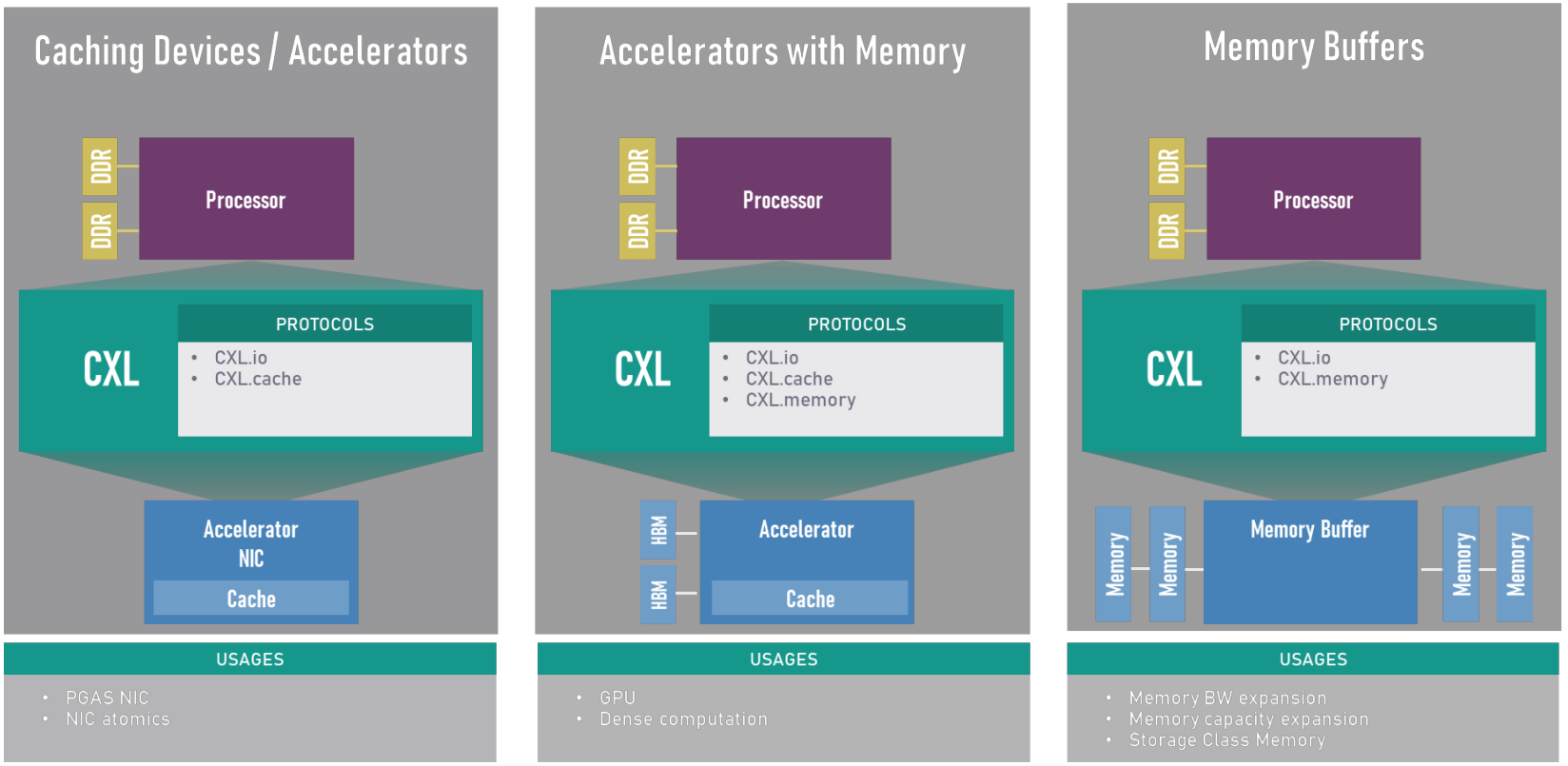
FPGA-based Near-Memory Acceleration of Modern Data-Intensive Applications

Gagandeep Singh[◇] **Mohammed Alser**[◇] **Damla Senol Cali**[⋈]
Dionysios Diamantopoulos[▽] **Juan Gómez-Luna**[◇]
Henk Corporaal[⋆] **Onur Mutlu**[◇⋈]

[◇]*ETH Zürich* [⋈]*Carnegie Mellon University*
[⋆]*Eindhoven University of Technology* [▽]*IBM Research Europe*

Compute Express Link (CXL)

- Compute Express Link (CXL) is an open industry standard interconnect offering high-bandwidth, low-latency connectivity between host processor and devices such as accelerators, memory buffers, and smart I/O devices



https://www.computeexpresslink.org/_files/ugd/0c1418_14c5283e7f3e40f9b2955c7d0f60bebe.pdf

NVIDIA Grace CPU and Grace Hopper

NEWSROOM MULTIMEDIA EXECUTIVE BIOS MEDIA CONTACTS IN THE NEWS

Press Release

Top Global Systems Makers Accelerate Adoption of NVIDIA Grace and Grace Hopper

Atos, Dell Technologies, GIGABYTE, Hewlett Packard Enterprise, Inspur, Lenovo and Supermicro Join First Wave Planning NVIDIA Grace-Powered HGX Systems for HPC and AI

Monday, May 30, 2022

NVIDIA today announced that a range of the world's leading computer makers are adopting the new NVIDIA Grace™ superchips to create the next generation of servers turbocharging AI and HPC workloads for the exascale era.

Atos, Dell Technologies, GIGABYTE, HPE, Inspur, Lenovo and Supermicro are planning to deploy servers built with the NVIDIA Grace CPU Superchip and NVIDIA Grace Hopper™ Superchip.

All these new systems benefit from the just-announced Grace and Grace Hopper designs in the NVIDIA HGX™ platform, which provide manufacturers the blueprints needed to build systems that offer the highest performance and twice the memory bandwidth and energy efficiency of today's leading data center CPU.

"As supercomputing enters the era of exascale AI, NVIDIA is teaming up with our OEM partners to enable researchers to tackle massive challenges previously out of reach," said Ian Buck, vice president of Hyperscale and HPC at NVIDIA. "Across climate science, energy research, space exploration, digital biology, quantum computing and more, the NVIDIA Grace CPU Superchip and Grace Hopper Superchip form the foundation of the world's most advanced platform for HPC and AI."

https://nvidianews.nvidia.com/news/top-global-systems-makers-accelerate-adoption-of-nvidia-grace-and-grace-hopper

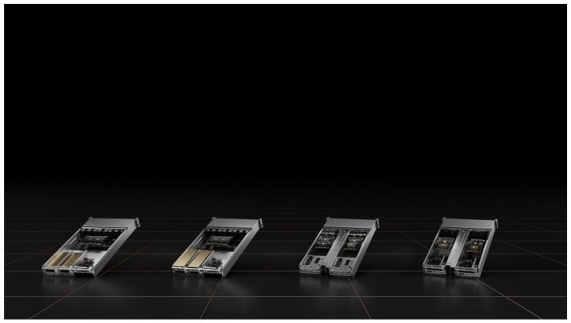
NEWSROOM MULTIMEDIA EXECUTIVE BIOS MEDIA CONTACTS IN THE NEWS

Press Release

Taiwan's Tech Titans Adopt World's First NVIDIA Grace CPU-Powered System Designs

New Class of Data Center Systems for Digital Twins, AI, High Performance Computing, Cloud Graphics and Gaming to Come From ASUS, Foxconn Industrial Internet, GIGABYTE, QCT, Supermicro, Wiwynn

Monday, May 23, 2022



COMPUTEX -- NVIDIA today announced that Taiwan's leading computer makers are set to release the first wave of systems powered by the NVIDIA Grace™ CPU Superchip and Grace Hopper Superchip for a wide range of workloads spanning digital twins, AI, high performance computing, cloud graphics and gaming.

Dozens of server models from ASUS, Foxconn Industrial Internet, GIGABYTE, QCT, Supermicro and Wiwynn are expected starting in the first half of 2023. The Grace-powered systems will join x86 and other Arm-based servers to offer customers a broad range of choice for achieving high performance and efficiency in their data centers.

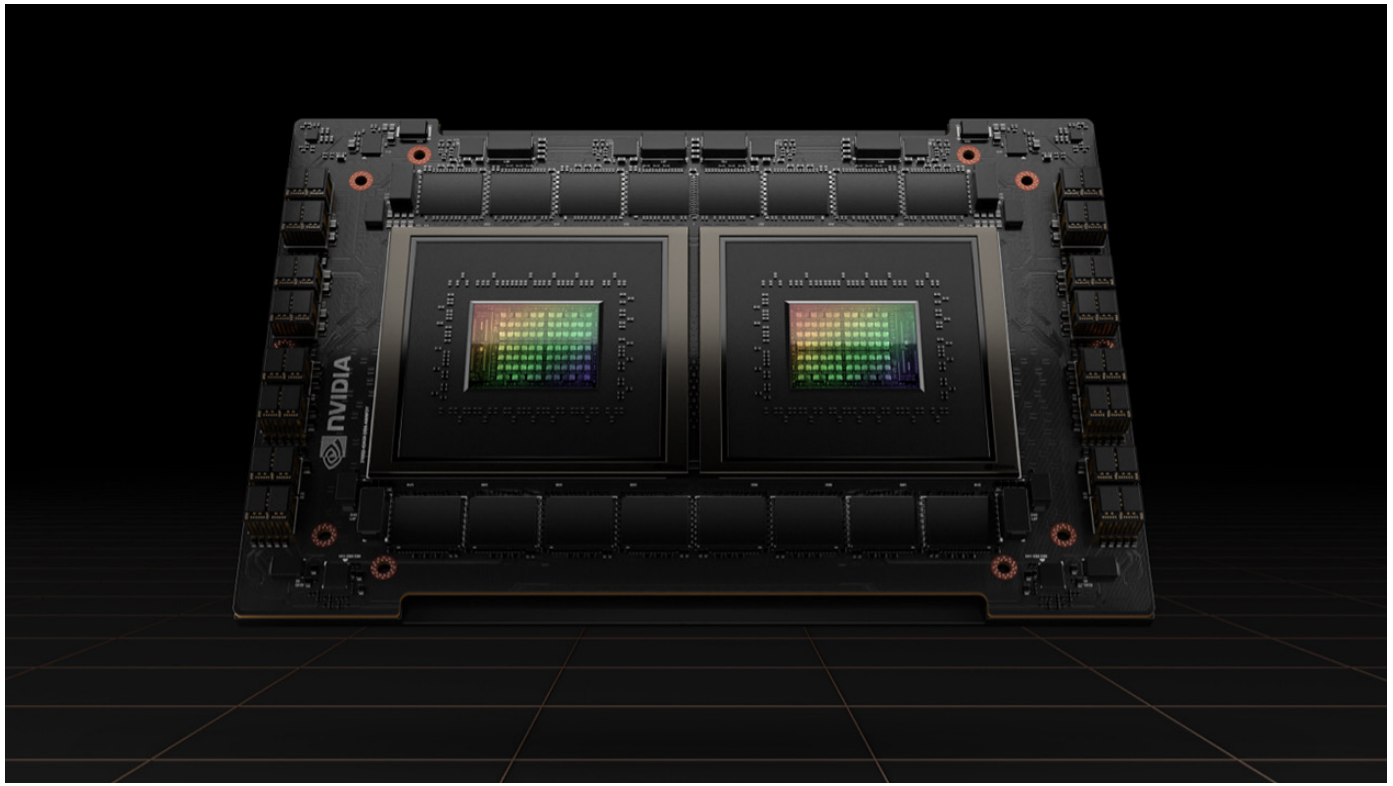
"A new type of data center is emerging — AI factories that process and refine mountains of data to produce intelligence — and NVIDIA is working closely with our Taiwan partners to build the systems that enable this transformation," said Ian Buck, vice president of Hyperscale and HPC at NVIDIA. "These new systems from our partners, powered by our Grace Superchips, will bring the power of accelerated computing to new markets and industries globally."

The coming servers are based on four new system designs featuring the Grace CPU Superchip and Grace Hopper Superchip, which NVIDIA announced at its two most recent GTC conferences. The 2U form factor designs provide the blueprints and server baseboards for original design manufacturers and original equipment manufacturers to quickly bring to market systems for the NVIDIA CGX™ cloud gaming, NVIDIA OVX™ digital twin and the NVIDIA HGX™ AI and HPC platforms.

https://nvidianews.nvidia.com/news/taiwans-tech-titans-adopt-worlds-first-nvidia-grace-cpu-powered-system-designs

NVIDIA Grace CPU Superchip

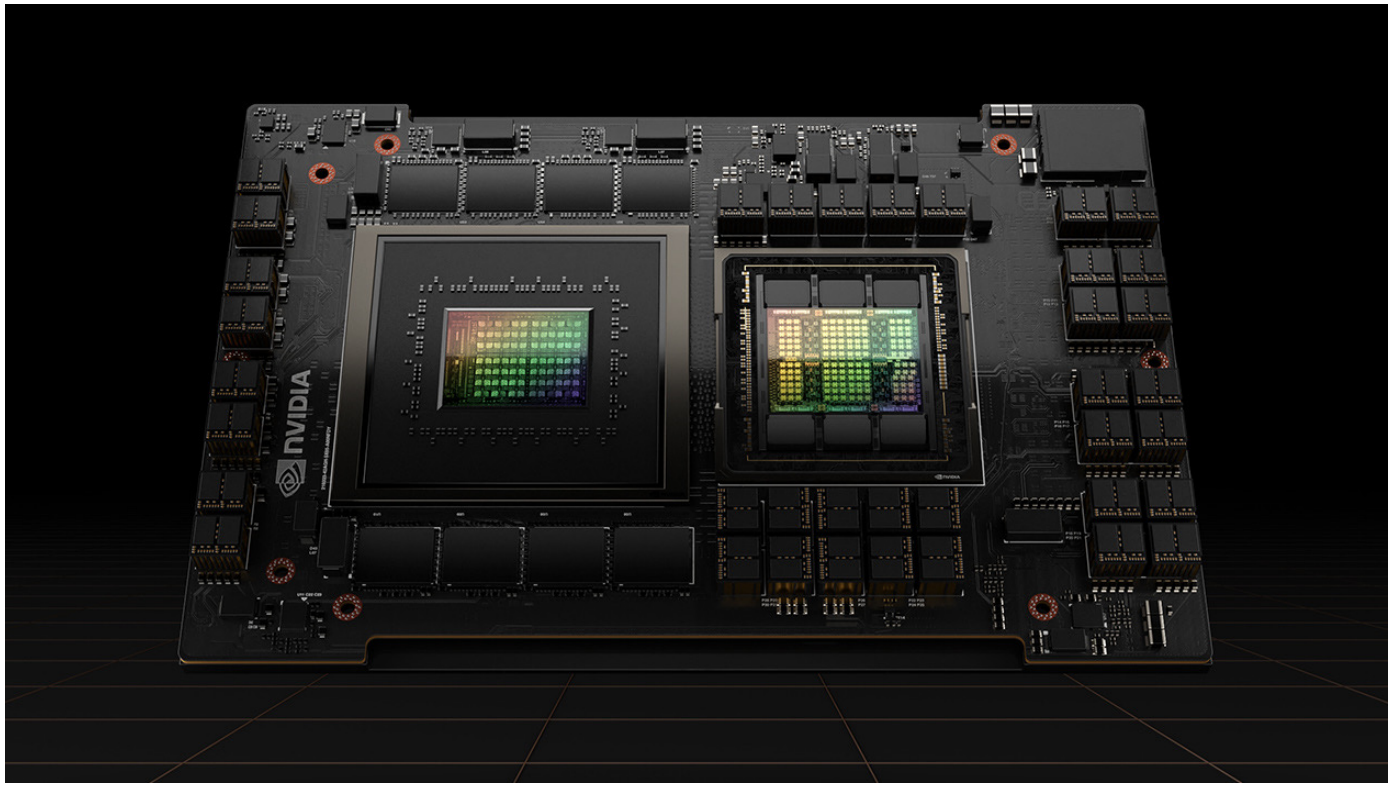
- 144 ARM v9 CPU cores
- LPDDR5x memory with ECC, 1 TB/s total bandwidth
- 900 GB/s coherent interface (7x faster than PCIe Gen 5)



https://www.nvidia.com/en-us/data-center/grace-cpu/

NVIDIA Grace Hopper Superchip

- CPU + GPU
  - Grace CPU + Hopper GPU
- 900 GB/s coherent interface (7x faster than PCIe Gen 5)



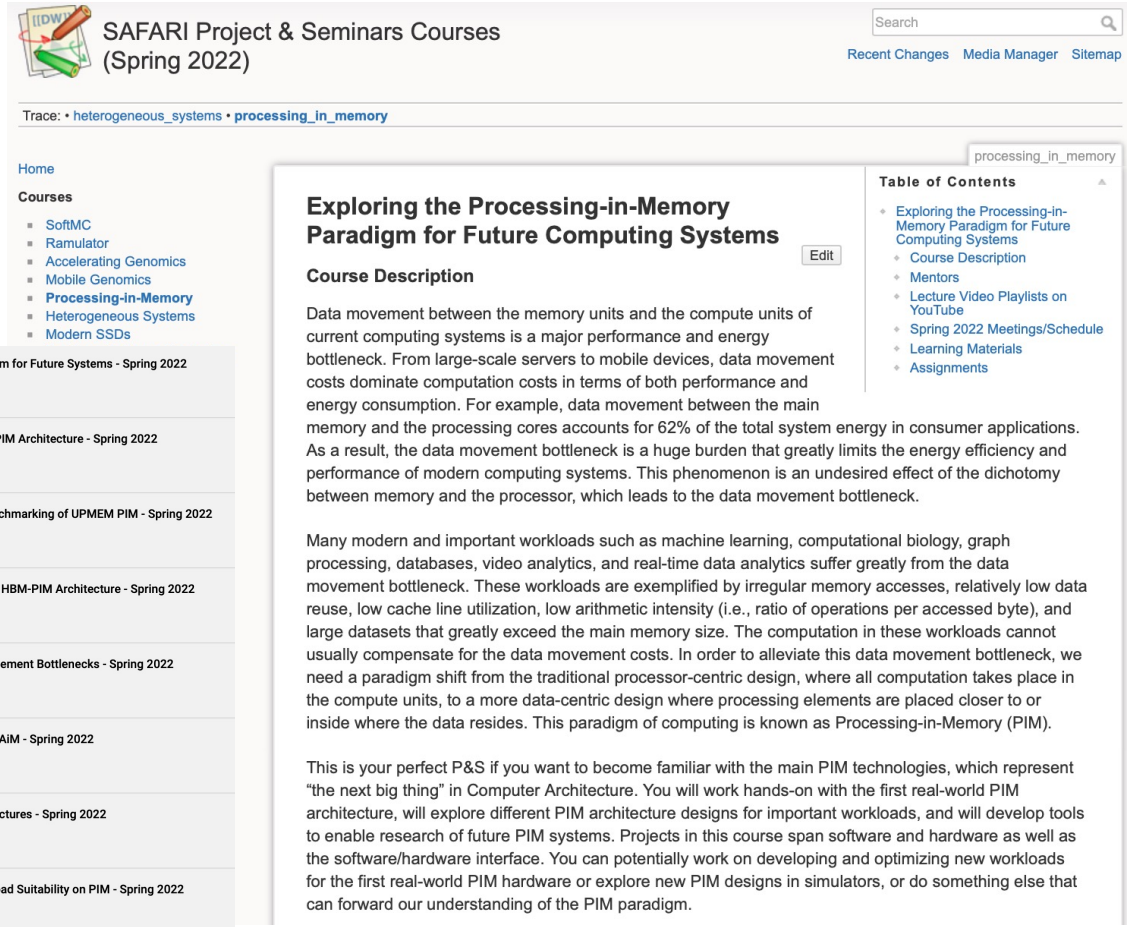
https://www.nvidia.com/en-us/data-center/grace-cpu/

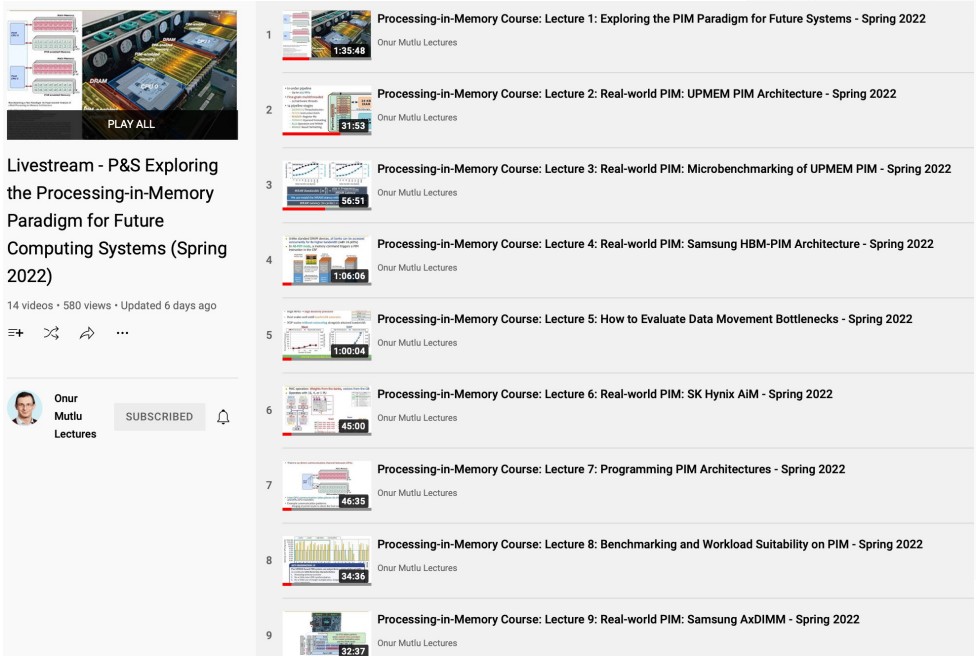
Collaborative Computing: Key Takeaways

- Possibility of having several devices collaborating on the same workload
- And having the most appropriate cores for each workload, exploiting heterogeneity
- Easier programming with Unified Memory or Shared Virtual Memory
- CPU-GPU memory coherence and system-wide atomic operations since NVIDIA Pascal and HSA
  - Fine-grain collaboration

Processing-in-Memory Course (Spring 2022)

- Short weekly lectures
- Hands-on projects



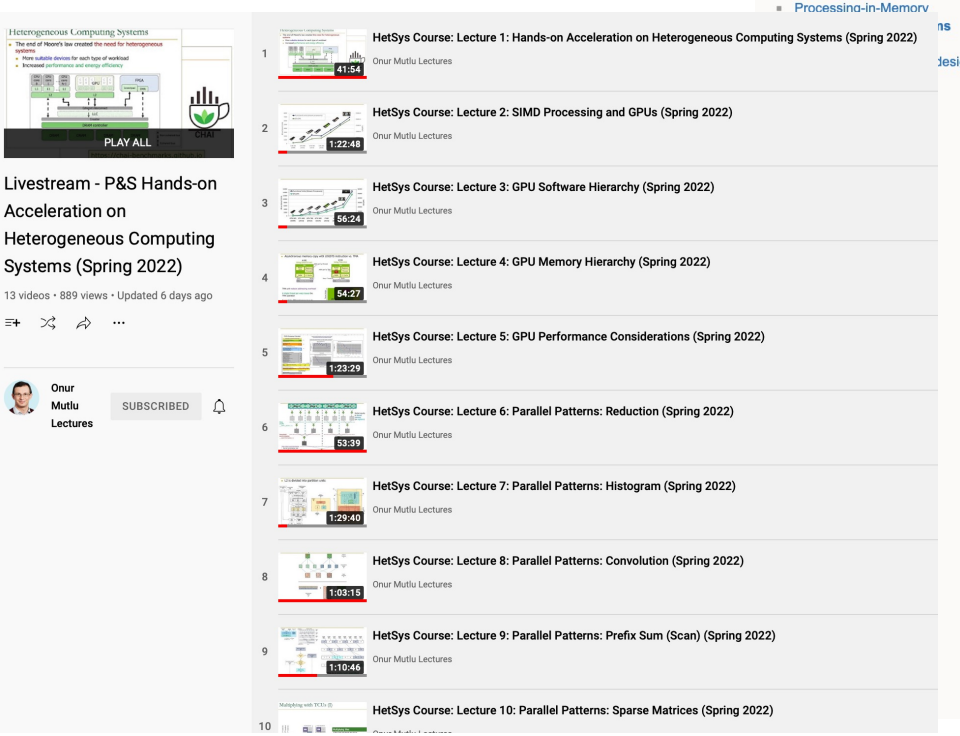


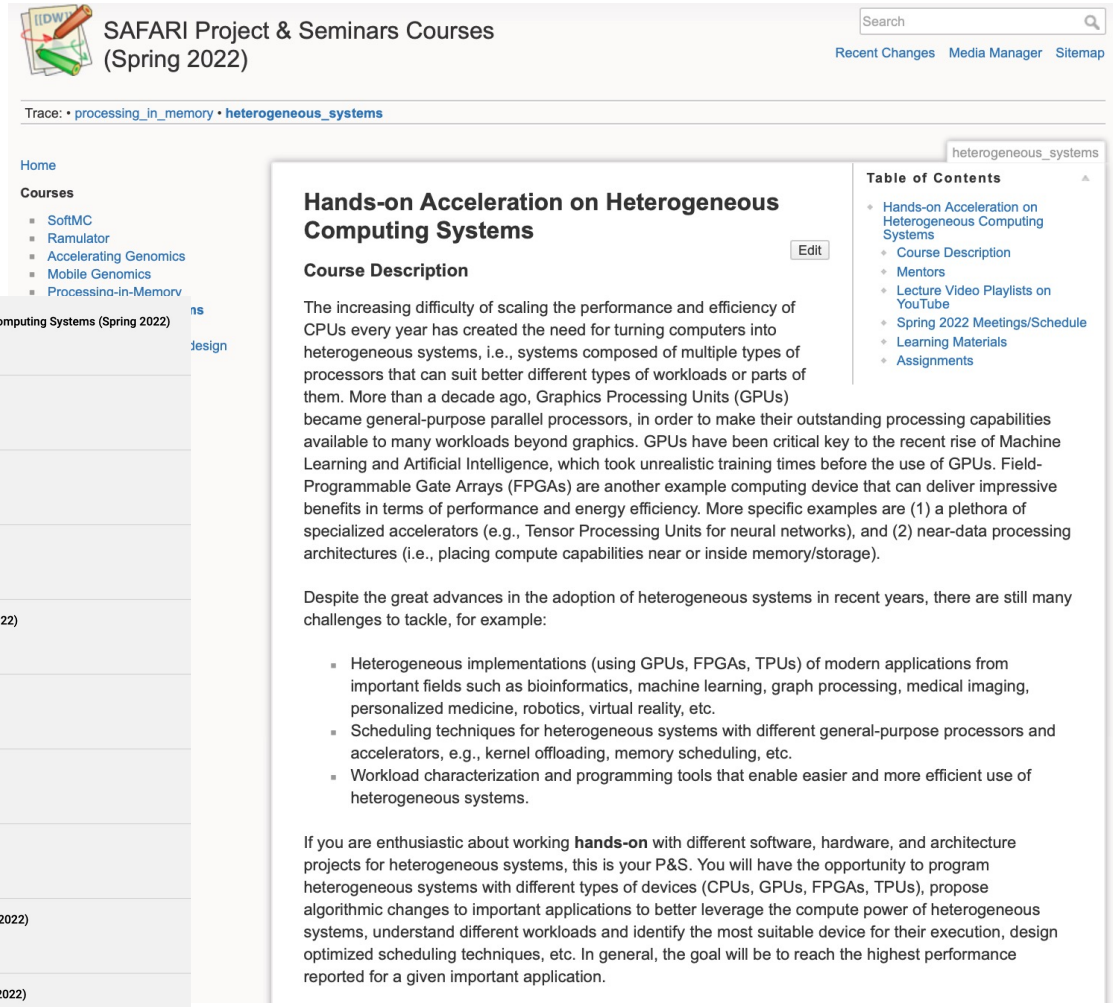
https://youtube.com/playlist?list=PL5Q2soXY2Zi-0NK1C5vi2Zx9nmE_3-cKN

https://safari.ethz.ch/projects_and_seminars/spring2022/doku.php?id=processing_in_memory

Heterogeneous Systems Course (Spring 2022)

- Short weekly lectures
- Hands-on projects





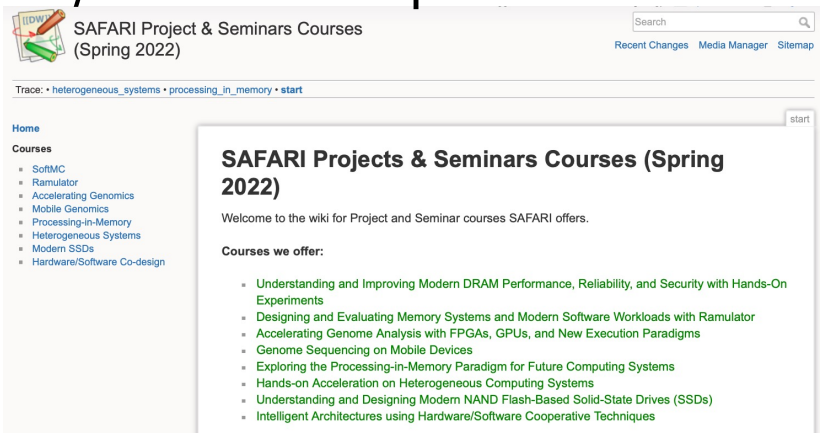
https://youtube.com/playlist?list=PL5Q2soXY2Zi9XrgXR38IM_FTjmY6h7Gzm

https://safari.ethz.ch/projects_and_seminars/spring2022/doku.php?id=heterogeneous_systems

More P&S Courses: SSDs, Memory, Bioinformatics…

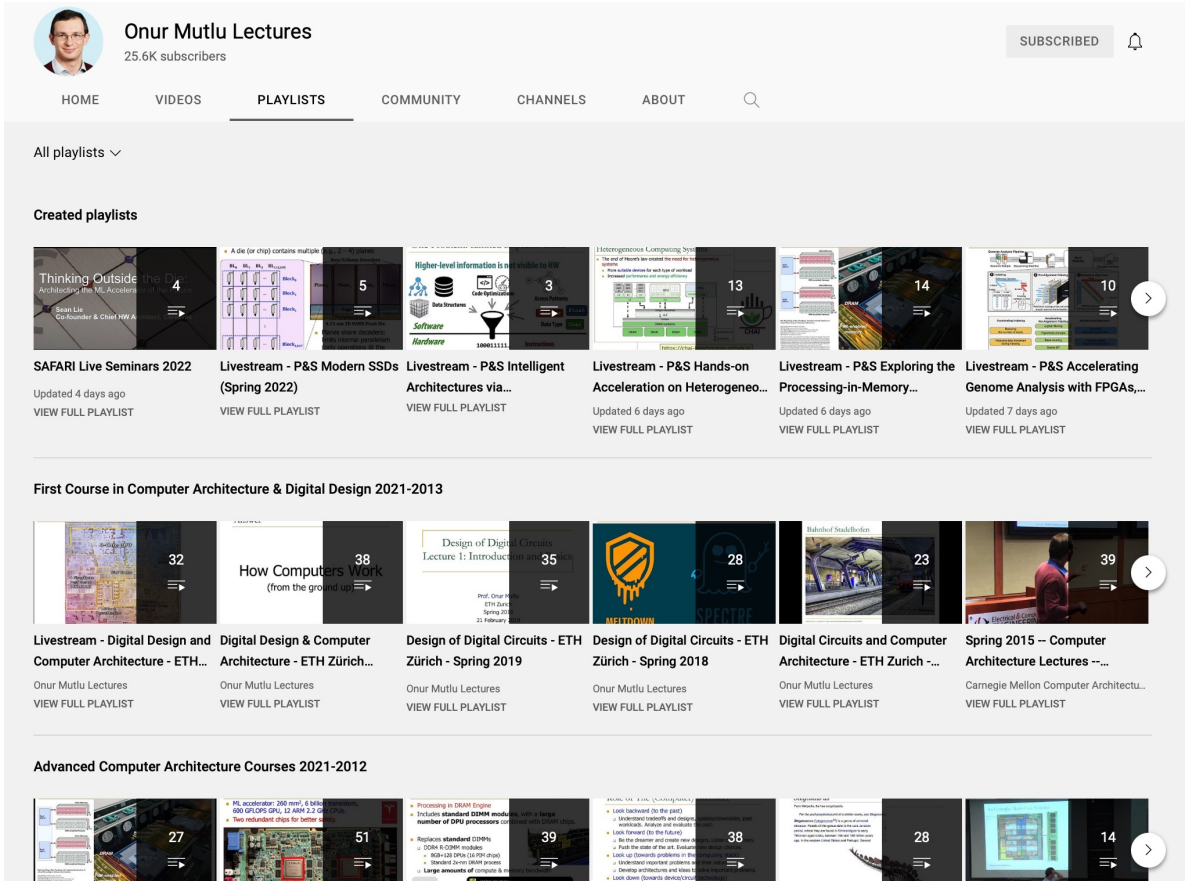
- Understanding and Improving Modern DRAM Performance, Reliability, and Security with Hands-On Experiments
- Designing and Evaluating Memory Systems and Modern Software Workloads with Ramulator
- Accelerating Genome Analysis with FPGAs, GPUs, and New Execution Paradigms
- Genome Sequencing on Mobile Devices
- Understanding and Designing Modern NAND Flash-Based Solid-State Drives (SSDs)
- Intelligent Architectures using Hardware/Software Cooperative Techniques

https://safari.ethz.ch/projects_and_seminars/spring2022/doku.php?id=start



More Resources: Onur Mutlu Lectures

- All P&S courses
- Digital Design and CompArch course
- Advanced CompArch course
- Seminar in CompArch



https://www.youtube.com/c/OnurMutluLectures/playlists

P&S Heterogeneous Systems

Collaborative Computing

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich

Fall 2022

16 January 2023